

Smoldyn

A Spatial Stochastic Simulator

Version 1.54, © March 2004

Steven Andrews

Part I: User manual

| | |
|--|-----------|
| 1. Overview of <i>Smoldyn</i> | 2 |
| 2. The Configuration File | 2 |
| 2.1 Starting <i>Smoldyn</i> | 2 |
| 2.2 Configuration file format | 3 |
| 2.3 Configuration file statements | 4 |
| 2.4 Runtime commands | 10 |
| 2.5 Sample configuration file | 13 |
| 3. Algorithm details | 14 |
| 3.1 Binding and unbinding radii | 15 |
| 3.2 Time steps | 17 |
| 3.3 Reactions near surfaces | 19 |
| 4. Tests of <i>Smoldyn</i> | 20 |
| 4.1 Initial test: <i>bounce</i> | 20 |
| 4.2 Diffusion rates: <i>diff</i> | 21 |
| 4.3 Zeroth order reaction rates: <i>zeroreact</i> | 22 |
| 4.4 Unimolecular reaction rates: <i>unireact</i> | 22 |
| 4.5 Bimolecular reactions, different reactants: <i>reactAB</i> | 23 |
| 4.6 Bimolecular reactions, same reactant: <i>reactAA</i> | 23 |
| 4.7 Reactions near walls: <i>reactW</i> | 24 |
| 4.8 Reversible reactions: <i>equil</i> | 24 |
| 4.9 Simple reaction networks: <i>lotvolt</i> | 25 |
| 5. Copyright and Citation | 25 |

Part II: *Smoldyn* Code documentation

| | |
|--|-----------|
| 1. Include files | 26 |
| 2. Constants, macros, and global variables | 27 |
| 3. Local variables | 29 |
| 4. Structures, allocation, and freeing routines | 30 |
| 5. Reaction parameter calculation routines | 40 |
| 6. Initialization routines | 42 |
| 7. Simulation routines | 43 |
| 8. Output routines | 46 |
| 9. Command interpreter routines | 47 |
| 10. Simulation management routines and main() | 51 |
| 11. <i>Smoldyn</i> modifications | 53 |

1. Overview of *Smoldyn*

Smoldyn is a computer program which simulates the dynamics of reaction-diffusion systems on a microscopic scale. While the code has been written to be platform independent, most recent development has been done on a Macintosh, running OS 10.2.1. It has also been compiled on a Linux system. OpenGL support (which is standard with most modern operating systems) is required for graphical output.

Molecules in the simulation are represented individually, diffuse within a rectangular volume by Brownian motion, and undergo simple chemical reactions. The timescale considered is short enough that diffusion within the simulation volume is explicitly modeled using exact molecular positions, but is long enough that momenta and molecular orientations can be assumed to take on average values. This is often called the Smoluchowski level of approximation (and hence the name of the program). A further approximation is that molecules do not occupy volume. Time steps are synchronous, alternating steps in which molecules diffuse and those in which they react. Space is defined as an arbitrary dimensional rectangular volume which has fixed walls. The walls can be reflective, absorbing, or periodic. Chemical reactions may be zeroth order, unimolecular, or bimolecular, and may have any number of products. While it is not currently possible to include surfaces or other structures in the system other than the walls that bound the simulation space, it is possible to account for fixed volumes that are excluded from the simulation volume. The algorithms for *Smoldyn* are described in a research paper written by Dennis Bray and myself titled "Stochastic simulation of chemical reactions with spatial resolution and single molecule detail." The paper has been submitted to the *Journal of Physical Chemistry* and can be provided upon request.

Smoldyn was written for computational biologists and computational chemists. While a significant amount can be done with the program without an understanding of how the algorithms work, this approach can easily lead to misleading results. Also, *Smoldyn* does not, and cannot hope to, include all the functionality that might be desired. Instead, it was written with a framework that allows extensions to be made relatively easily, although this still requires a moderate commitment and familiarity with the C language.

2. The Configuration File

2.1 Starting *Smoldyn*

Before running *Smoldyn*, the parameters for a simulation need to be written in a text format configuration file and saved to disk. Some sample configuration files should have been included with the executable program, one of which is presented in section 2.5 of this documentation. When *Smoldyn* is run, the program asks for the name of the configuration file and for a few additional parameters; then it loads the file, runs the simulation, and terminates. The only user input that is processed during the simulation is for simple graphics manipulation and simulation pausing.

The name of the configuration file is typically given to *Smoldyn* after the program is running, when it asks for the name of a configuration file. At the text prompt, type in the

configuration file name, including file path information. Spaces are allowed in the file name or file path. Here are some examples, using a configuration file called `config` and Macintosh file path notation:

| | |
|---|---|
| config is in same directory as <i>Smoldyn</i> : | <code>config</code> |
| config is in a subdirectory of where <i>Smoldyn</i> is: | <code>:folder:config</code> |
| Absolute path name, starting from the volume: | <code>Mac HD:files:folder:config</code> |

It should also be possible to have a relative file path that ascends the file tree, as well as descending the tree, although I have not figured out how to do this with the Macintosh file system yet. Standard file path notation should also work with PCs or with Linux, but have not been tested.

After the configuration file name is entered, *Smoldyn* asks for runtime flags, where options are the characters 'q', 'p', '-', or a combination of characters. 'q' is used for quiet operation, in which no diagnostics or parameters are displayed, and is minimally useful. 'p' indicates that the simulation should be set up and that all diagnostics and parameters should be calculated and displayed, but then the actual simulation should not be run. This is useful for choosing appropriate parameters for complex simulations. A '-' is used to indicate that neither the 'q' nor 'p' options are desired.

It should also be possible to run *Smoldyn* from a Unix style command line, in which case the name of the configuration file and the runtime flags can be entered on the same command line, thus removing the need for text entry during program execution. In this case, the order of the file name and the flags is unimportant and any runtime flags need to be preceded by a '-'. I have not figured out how to do this yet using the Macintosh system.

With the exception of graphics, which cannot be saved, all output from *Smoldyn* is saved to text files, allowing their analysis with a wide variety of other software. These output file names are declared in the configuration file, as explained below. These output files can be saved in the same file folder as the configuration file, or in a sub-directory of that folder.

2.2 Configuration file format

The design of a simulation can be broken down into two portions. One portion includes the parameters of the physical system, including its shape, the molecules that are present, and the reactions that take place. These parameters are entered in the configuration file using a variety of statements and are simulated using the core program. Here are some examples of statements:

| | |
|-----------------------------------|----------------------------------|
| <code>dim 3</code> | 3 dimensional system |
| <code>high_wall 0 100 r</code> | properties of a system boundary |
| <code>time_start 0</code> | starting time for the simulation |
| <code>mol 10 CheY 50 50 50</code> | creation of 10 CheY molecules |

The other portion of a simulation is the action of the experimenter, which includes measurements of the system and external perturbations to the system. These actions are listed, also in the configuration file, with a series of commands with execution times; they are not considered by *Smoldyn* until they are supposed to happen. When a command is supposed to be executed, *Smoldyn* processes it with a runtime command interpreter, which is an auxiliary portion of the program that is designed to be easily modified. Here are some examples of commands:

```
cmd e ifno asp stop      conditional command, run at every iteration
cmd @ 20 molcount outfile  molecules are counted and recorded at time 20
```

Simulation parameters need to be entered using the formats shown in the table given below. Formatting errors should be caught by *Smoldyn*, causing a program termination and, hopefully, a useful error message. On each line of input, *Smoldyn* reads a word that tells what the line contains and then reads the number of items that it expects. Both too few items and too many items cause errors, although it is always possible to add comments to the end of a line using a number sign. In many cases, lines may be entered in any order, although some basic definitions need to be entered near the top of the file. Default values are used for parameters that are not defined in the configuration file. While many instructions can only be entered once, such as the system dimensionality, others can be entered multiple times, such as lines to define various collections of molecules.

Reactions are entered as a block of definitions, beginning with the word “start_reaction” and ending with “end_reaction”, between which only instructions that are relevant to reactions are allowed. Reactions are stored internally with two fundamental parts: a reactant table and a list of reactions, including both rates and products. The reactant table associates reactants with numbered reactions but does not give further details about them; in effect it is only the set of species on the left side of the arrows in a list of reactions. The reaction list includes rate constants and lists of products; in effect it is the set of species on the right side of the arrows. This structure is used in the reaction section of the input file as well. A confusing aspect of reactions is that reversible reactions (and some so-called continuation reactions) require a parameter from which *Smoldyn* can figure out where to place multiple reaction products to prevent immediate recombination. This issue is explained in section 3 of this documentation.

Since *Smoldyn* does not use any particular set of units, it is up to the user to make sure units are consistent. Some useful conversion factors are:

$$10^{-6} \text{ cm}^2 \text{ s}^{-1} = 10^{-10} \text{ m}^2 \text{ s}^{-1} = 100 \text{ } \mu\text{m}^2 \text{ s}^{-2} = 0.1 \text{ } \mu\text{m}^2 \text{ ms}^{-1}$$

$$1 \text{ M}^{-1} \text{ s}^{-1} = 10^{-3} \text{ m}^3 \text{ mol}^{-1} \text{ s}^{-1} = 1.66 \times 10^{-27} \text{ m}^3 \text{ s}^{-1} = 1.66 \times 10^{-18} \text{ } \mu\text{m}^3 \text{ ms}^{-1}.$$

2.3 Configuration file statements

Statements about the configuration file

text

Comment. A '#' symbol indicates that the rest of the line is a comment.

`read_file filename`

Read some other configuration file, returning to the present one when that one has been read.

`end_file`

End of configuration file. This line is optional, as *Smoldyn* can also just read until the file ends.

Definition of system parameters

`dim dim`

Dimensionality of the system. Must be at least one, and is typically between 1 and 3. Larger numbers are permitted as well.

`names name1 name2 ... namen`

Names of the types of molecules present in the system. Standard naming conventions are followed, in that the name should start with a letter and spaces are not permitted.

`diff name float`

Isotropic diffusion constant of molecule type *name*. Default value is 0.

`diffm name float0 float1 ... floatdim*dim-1`

Square root of diffusion matrix of *name* (the dot product of this matrix and itself is the anisotropic diffusion matrix). The matrix has dim^2 terms (*dim* is the system dimensionality), listed row by row of the matrix; the matrix is supposed to be symmetric. If this line is not entered, isotropic diffusion is assumed, which leads to a faster runtime. While a matrix is used for diffusion if one is given, the value stored with `diff` is used for reaction rate calculations. If `diff` is not entered, the trace of the square of this matrix, divided by the system dimensionality, is used as a proxy for the isotropic diffusion coefficient to allow reaction rates to be estimated. This line is most useful for restricting diffusion to a plane or a line, in which case the square root of the diffusion coefficient is given for each diagonal element of the matrix where there is diffusion and 0s are placed on diagonal elements for axes where diffusion is not possible, as well as on off-diagonal elements.

`low_wall dim pos type`

Creates a lower boundary for the simulation volume. This wall is perpendicular to the dimension *dim* such that all locations between *pos* and the position of the corresponding upper boundary are considered to be within the simulation volume. The type of wall is given in *type*, which should be one of four single letter codes: 'r' means a reflecting wall, 'p' means a periodic wall (also called wrap-around or toroidal), 'a' means an

absorbing wall, and 't' means a transparent wall. Transparent walls imply an unbounded system and may lead to slow simulations.

high_wall dim pos type

Identical to the definition for *low_wall*, although this creates the upper boundary for the simulation volume.

max_mol int

Maximum possible number of molecules for which memory should be allocated. The simulation terminates if more molecules are required than are allocated initially. Note that most reactions require a few extra molecule spaces for processing.

mol nmol name pos₀ pos₁ ... pos_{dim-1}

Simulation starts with *nmol* type *name* molecules at location *pos*. Each of the *dim* elements of the position may be a number to give the actual position of the molecule or molecules, or the letter 'u' to indicate that the position for each molecule should be a random value between the bounding walls, chosen from a uniform density.

Simulation performance statements

rand_seed int

Seed for random number generator. If this line is not entered, the current time is used as a seed, producing different sequences for each run.

accuracy float

A parameter that determines the quantitative accuracy of the simulation, on a scale from 0 to 10. Low values are less accurate but run faster. Default value is 10, for maximum accuracy. When accuracy is 0, bimolecular reactions are only checked for pairs of reactants that are both within the same virtual box; with higher accuracy values, reactants in nearest neighboring boxes are considered as well, and then when accuracy is 10, reactants in all types of neighboring boxes are checked.

molperbox float

Virtual boxes are set up initially so the average number of molecules per box is no more than this value. The default value is 5. *boxsize* is an alternate way of entering comparable information.

boxsize float

Rather than using *molperbox* to specify the sizes of the virtual boxes, *boxsize* can be used to request the width of the boxes. The actual box volumes will be no larger than the volume calculated from the width given here.

Graphical display statements

graphics *str*

Type of graphics to use during the simulation. Currently, the only options are 'none' and 'opengl'. If this line is not entered, no graphics are shown.

graphic_iter *int*

Number of time steps that should be run between each update of the graphics. Default value is 1. This line is irrelevant if graphics aren't being used.

frame_thickness *int*

Thickness of the frame that is drawn around the simulation volume, in points. Default value is 2. This line is irrelevant if graphics aren't being used.

display_size *name int*

Size of molecule of type *name* for display to the graphical output. The default value is 3, indicating that each molecule is displayed with a small square; 0 indicates that a molecule should not be displayed and larger numbers yield larger squares.

color *name red green blue*

Red, green, and blue values for displaying molecules of type *name*. Each value should be between 0 and 1. Default values are 0 for each parameter, which is black. Some useful colors: black is 0 0 0, brown is 0.6 0.6 0, red is 1 0 0, orange is 1 0.7 0, yellow is 0.8 0.9 0, green is 0 1 0, blue is 0 0 1, violet is 0.6 0 0.6, grey is 0.4 0.4 0.4, white is 1 1 1, and blue-green is 0 0.6 0.5.

Simulation time statements

time_start *float*

Starting point for simulated time.

time_stop *float*

Stopping time of simulation, using simulated time. The simulation continues past the time_stop value by less than one time step.

time_step *float*

Time step for the simulation. Longer values lead to a faster runtime, while shorter values lead to higher accuracy. Also, longer values lead to bimolecular reactions that behave more as though they are activation limited, rather than diffusion limited.

time_now *float*

Another starting time of simulation. Default value is equal to `time_start`. If this time is before `time_start`, the simulation starts at `time_start`; otherwise, it starts at `time_now`.

Statements about the runtime command interpreter

`output_root` *str*

Root of path where text output should be saved. Spaces are permitted. Output files are saved in the same folder as the configuration file, modified by this string. See the description for `output_files`. Make sure that the destination folder has been created and that the string is terminated with a colon (and started with a colon if needed).

`output_files` *str₁ str₂ ... str_n*

Declaration of filenames that can be used for output of simulation results. Spaces are not permitted in these names. Any previous files with these names will be overwritten. The path for these filenames starts from the configuration file and may be modified by a root given with `output_root`. For example, if the configuration file was called with `:folder:config.txt` and `output_root` was not used, then the output file `out.txt` will appear in the folder `folder` too. If the configuration file was called with `:folder:config.txt` and the output root was given as `results:`, then the output file goes to the `results` sub-folder of the folder `folder`. The filename “`stdout`” results in output being sent to the standard output. In most cases, it is also permissible to not declare filenames, in which case output is again sent to the standard output.

`output_file_number` *int*

Starting number of output file name. The default is 0, meaning that no number is appended to a name (e.g. the file name `out.txt` is saved as `out.txt`). A value larger than 0 leads to an appended file name (if 1 is used, then `out.txt` is actually saved as `out_001.txt`). Note that the command `incrementfile` increments the file number before it runs the rest of the command.

`max_cmd` *int*

Maximum length of command queue. Default value is 10.

`cmd b,a,e` *string*

`cmd @` *time string*

`cmd n` *int string*

`cmd i` *on off dt string*

Declaration of a command to be run by the run-time interpreter, where the final portion labeled *string* is the actual command. The character following `cmd` is the command type, which may be ‘b’ for before the simulation, ‘a’ for after the simulation, ‘e’ for every time step during the simulation, ‘@’ for a single command execution at time *time*, ‘n’ for every

n 'th iteration of the simulation, or 'i' for a fixed time interval. For type 'i', the command is executed over the period from *on* to *off* with intervals of at least dt (the actual intervals will only end at the times of simulation time steps). See section 2.4 for the commands that are available. Note that command execution times may not be exactly correct because of round-off errors.

Reaction definitions

`start_reaction`

Start of reaction definition. Between this instruction and "end_reaction", all lines need to pertain to this order of reaction. It is permissible to list reactions of the same order in multiple blocks, provided that only the first block includes a `max_rxn` statement and that sufficient reactions are declared with that statement.

`order int`

Order of the reactions being declared (0, 1, or 2).

`max_rxn max_rxn`

Maximum number of reactions that will be declared of the given order.

`reactant $r_0 r_1 \dots r_{nrxn-1}$`

`reactant name $r_0 r_1 \dots r_{nrxn-1}$`

`reactant name1 + name2 $r_0 r_1 \dots r_{nrxn-1}$`

Declaration of reactants and reaction names for zeroth order, unimolecular, and bimolecular reactions, respectively. The listed molecule names are the reactants and the following strings are the respective reaction names. Note that there are spaces before and after the '+' symbol.

`rate r rate`

Reaction rate constant for reaction called r . Units for the reaction rate constant are $(\text{volume})^{\text{order}-1}$ times inverse time. These rates are converted by the program into probabilities or binding radii. To enter the simulation parameters directly, use `rate_internal`.

`rate_internal r float`

Internal value for reaction rate information, which can be used to override the internal rates that are calculated from the `rate` entry. For zeroth order reactions, this is the expectation total number of reactions per time step; for unimolecular reactions, this is the reaction probability per time step for each reactant molecule; and for bimolecular reactions, this is the binding radius.

`product r name + name + ... + name`

List of products for reaction r . Note that there are spaces before and after each '+' symbol.

product_param r i

product_param r p, x, r, b, q, y, s *float*

product_param r o, f *prod_name pos₀ pos₁... pos_{dim-1}*

Parameters for the initial placement of products of reaction r . A product parameter also affects the binding radius of the reverse reaction. These are explained in section 3. In the first format, a type of 'i' indicates that the reverse reaction is ignored for calculations. The second format uses one of the type letters shown: 'p' and 'q' are geminate rebinding probabilities, 'x' and 'y' are maximum geminate rebinding probabilities, 'r' and 's' are ratios of unbinding to binding radii, and 'b' is a fixed unbinding radius. The third format yields products that have a fixed relative orientation, which is either randomly rotated with 'o', or not rotated with 'f'. In the absence of better information, a useful default parameter type is either 'x' or 'y', with a value of about 0.2.

end_reaction

End of reaction definition. Reaction instructions are no longer recognized but other simulation instructions are.

2.4 Runtime commands

Commands are stored in a queue, which is checked and executed just before the simulation starts, during the simulation, and just after it ends. All commands are declared in the configuration file using one of the forms shown above, where the final string is the actual command text. In some cases, the command text allows additional commands to be entered as well, allowing conditional expressions. Following is a list of possible command strings.

Simulation control commands

stop

Stop the simulation.

pause

This puts the simulation in pause mode. If opengl graphics are used, continuation occurs when the user presses the spacebar. When graphics are not used, the user is told to press enter.

File manipulation commands

overwrite *filename cmd*

Erase the output file called *filename* and then run command *cmd*.

incrementfile *filename cmd*

A new output file is created based upon the *filename*. The first time this is called the filename is appended with a “_001”, which is then incremented with subsequent calls to “_002”, and so on. These numbers precede any suffix on the filename.

Conditional commands

ifno *name cmd*

Run command *cmd* if no molecule of type *name* remains.

ifless *name num cmd*

Run command *cmd* if there are less than *num* molecules of type *name* remaining.

ifmore *name num cmd*

Run command *cmd* if there are more than *num* molecules of type *name*.

System manipulation commands

pointsource *name num pos₀ pos₁ ... pos_{dim}*

Create *num* new molecules of type *name* and at location *pos*.

killmol *name*

Kill all molecules of type *name*.

equilmol *name₁ name₂ prob*

Equilibrate these molecules. All molecules of type *name₁* and *name₂* will be randomly replaced with one of the two types, where type *name₂* has probability *prob*.

replacexyzmol *name pos₀ pos₁ ... pos_{dim-1}*

If there is a non-diffusing molecule at exactly position *pos*, it is replaced with one of type *name*. This command stops after one molecule is found.

modulatemol *name₁ name₂ freq shift*

Modulates molecules of types *name₁* and *name₂*, just like equilmol, but with a variable probability. Every time this command executes, any of the two types of molecules in the system are replaced with a molecule of type *name₁* with probability $\cos(freq * t + shift)$, where *t* is the simulation time, and otherwise with a molecule of type *name₂*.

react1 *name rxn*

All molecules of type *name* are instantly reacted, resulting in the products and product placements given by the unimolecular reaction named *rxn*. Note that *name* does not have to be the normal reactant for reaction *rxn*.

excludebox *xlo xhi*
excludebox *xlo xhi ylo yhi*
excludebox *xlo xhi ylo yhi zlo zhi*

This keeps all molecules from entering a rectanguloid box within the system volume. Use the first form for one dimension, the second for two dimensions, and the third for three dimensions. Molecules that start within the box can stay there, but any molecule that tries to diffuse into the box is returned to its location at the previous time step. This command needs to be run at every time step to work properly.

excludesphere *x rad*
excludesphere *x y rad*
excludesphere *x y z rad*

This keeps all molecules from entering a sphere within the system volume. Use the first form for one dimension, the second for two dimensions, and the third for three dimensions; the coordinates given are the sphere center and *rad* is the sphere radius. Molecules that start within the sphere can stay there, but any molecule that tries to diffuse into the sphere is returned to its location at the previous time step. This command needs to be run at every time step to work properly.

includeecoli

An *E. coli* shape is defined as a cylinder with hemispherical endcaps, where the long axis of the bacterium extends the length of the *x*-axis within the system walls and the radius of both the cylinder and the endcaps is half the spacing between the walls that bound the *y*-axis. This command moves any molecule that diffuses out of the *E. coli* shape back to its location at the previous time step, or to the nearest surface of the *E. coli* if it was outside at the previous time step as well. This command does not need to be run at every time step to work properly. This only works with a 3 dimensional system.

System observation commands

molcount *filename*

Each time this command is executed, one line of display is printed to the listed file, giving the time and the number of molecules for each molecular species. The ordering used is the same as was given in the names command.

listmols *filename*

This prints out the identity and location of every molecule in the system to the listed file name, using a separate line of text for each molecule.

listmols2 *filename*

This is very similar to `listmols` but has a slightly different output format. Each line of text is preceded by the “time counter”, which is an integer that starts at 1 and is incremented each time the routine is called. Also, the names of molecules are not printed, but instead the identity numbers are printed.

`listmols3 name filename`

This is identical to `listmols2` except that it only prints information about molecules of type *name*.

`molpos name filename`

This prints out the time and then the positions of all molecules of type *name* on a single line of text, to the listed filename.

`molmoments name filename`

This prints out the positional moments of the molecule type given to the listed file name. All the moments are printed on a single line of text; they are the number of molecules, the mean position vector (*dim* values), and the variances on each axis and combination of axes (*dim*² values).

`savesim filename`

This writes the complete state of the current system to the listed file name, in a format that can be loaded in later as a configuration file. Note that minor file editing is often desirable before simulating a file saved in this manner. In particular, the saved file will declare its own name as an output file name, which will erase the configuration file.

2.5 Sample configuration file

The following sample file executes a Lotka-Volterra reaction scheme, using parameters that are essentially the same as those used by Gillespie in his classic paper (*J. Phys. Chem.* 81:2340-2361, 1977). Using a vaguely ecological concept, R stands for rabbit and F stands for fox. The reactions are

```
R → 2 R
R + F → F
F → nothing
```

The simulation is run in three dimensions with periodic boundary conditions.

```
# Simulation file for Lotka-Volterra reaction
```

```
graphics opengl
graphic_iter 5
accuracy 5
# rand_seed 5
```

```

dim 3
names R F
max_mol 20000
molperbox 1

difc R 100
difc F 100
color R 1 0 0
color F 0 1 0
display_size R 2
display_size F 3

time_start 0
time_stop 100
time_step 0.001

low_wall 0 -100 p
high_wall 0 100 p
low_wall 1 -100 p
high_wall 1 100 p
low_wall 2 -10 p
high_wall 2 10 p
mol 1000 R u u u
mol 1000 F u u u

max_cmd 10
cmd b pause
cmd e ifno R stop
cmd e ifno F stop

start_reaction
order 1
max_rxn 2
reactant R Rmultiply      # R -> 2R
rate Rmultiply 10
product Rmultiply R + R
reactant F Fdie           # F -> 0
rate Fdie 10
end_reaction

start_reaction
order 2
max_rxn 1
reactant R + F Feat      # R+F -> 2F
rate Feat 8000
product Feat F + F
end_reaction

end_file

```

3. Algorithm details

3.1 Binding and unbinding radii

For every bimolecular reaction, *Smoldyn* has to calculate the correct binding radius from the reaction rate that is given in the configuration file. Also, for every reaction that leads to multiple products, *Smoldyn* has to determine the correct unbinding radius, using whatever product parameter is supplied, if any. While these binding and unbinding radii are well defined microscopic parameters (at least within the context of the analytical model system that is simulated), the meanings of the experimental rate constants, as well as those given in the configuration file, are not nearly as well defined. Instead, those rate constants depend on the conditions under which they were measured. *Smoldyn* accounts for this by attempting to guess the experimental conditions, using a process described here. If *Smoldyn*'s guess is correct, the simulated reaction rates should exactly match the experimental rates (not including edge effects, which are typically negligible unless one reactant is fixed at or near an edge).

The product parameters are:

Use these if reversible reactions were measured at equilibrium

- p probability of geminate reaction (\square).
- x maximum probability of geminate reaction (\square_{max}).
- r unbinding radius relative to binding radius (\square_u/\square_b).
- b fixed length unbinding radius (\square_u).

Use these if reversible reactions measured with all product removed as it was formed

- q probability of geminate reaction (\square).
- y maximum probability of geminate reaction (\square_{max}).
- s unbinding radius relative to binding radius (\square_u/\square_b).
- o fixed offset of products, rotationally randomized (\square_u).
- f fixed offset of products, not rotationally randomized (\square_u).
- i reaction is declared irreversible ($\square_u=0$).

In all cases, *Smoldyn* assumes that rate constants were measured using an effectively infinite amount of reactants that were started well mixed and that then were allowed to react until either an equilibrium was reached for reversible reactions, or a steady-state reaction rate was reached for irreversible reactions. Only in one of these cases is mass action kinetics correct and is the rate constant actually constant. The precise experimental assumptions are clarified with the following examples.



The rate constant is assumed to have been measured at steady state, starting with a well-mixed system of A and B. No product parameter is required. At steady-state, the simulation matches mass action kinetics.



There is no bimolecular reaction, so no binding radius is calculated. The default unbinding radius is 0, although it is possible to define a different one. If the product parameter is p, q, r, or s, an error is returned due to the lack of a binding radius. If it is not given or is i, x, or y, the unbinding radius is set to 0. If it is b, f, or o, the requested separation is used. At steady-state, the simulation matches mass action kinetics.

3. $A+B \rightleftharpoons C$

If the reversible parameter is p, x, b, or r, the forward rate constant is assumed to have been measured using just this system of reactions after the system had reached equilibrium. The product parameter is used to yield the correct probability of geminate recombination if possible, or the desired unbinding radius. In this case, the simulation matches mass action kinetics at equilibrium. If the product parameter is q, y, s, o, f, or i, then it is assumed that the forward rate constant was measured at steady-state and with all C removed as it was formed, thus preventing any geminate reactions. The unbinding radius is set as requested, using the binding radius if needed. In this case, the simulated forward reaction rate is higher than requested due to geminate rebindings.

4. $A+B \rightleftharpoons C \rightleftharpoons Y$

The second reaction is ignored for determining parameters for A+B. Instead, the first reaction is considered as though the rates were determined experimentally using just the system given in example 3. If the product parameter is p, x, r, or b, the simulated reaction rate for the forward reaction $A+B \rightleftharpoons C$ will be lower than the requested rate because there are fewer geminate reactions than there would be with the equilibrium system. Alternatively, it will be higher than the requested rate if the product parameter is q, y, s, o, f, or i, because there are some geminate reactions.

5. $X \rightleftharpoons A+B \rightleftharpoons C$

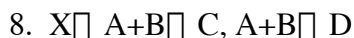
The binding radius for the second reaction is treated as in example 1, without consideration of the first reaction. The unbinding radius for the first reaction is found using the binding radius of the second reaction. Here, product parameters p and q are equivalent, x and y are equivalent, and r and s are equivalent. The actual reaction rate for the second reaction, found with a simulation, will be higher than the requested value due to geminate rebindings that occur after the dissociation of X molecules.

6. $X \rightleftharpoons A+B \rightleftharpoons C$

The $A+B \rightleftharpoons C$ binding and unbinding radii are treated as in example 3. Another unbinding radius is required for the first reaction, which is found as in example 5, using the binding radius from the second reaction. Mass action kinetics are not followed.

7. $X \rightleftharpoons A+B \rightleftharpoons C$

The binding radii and unbinding radii for each bimolecular reaction are found as in example 3, independent of the other bimolecular reaction. The simulated rates may be different from those requested because of differing unbinding radii.



The binding radii for the two bimolecular reactions are each found as in example 1. The unbinding radius for the first reaction cannot be determined uniquely, because the two forward reactions from $A+B$ are equivalent and are likely to have different binding radii. *Smoldyn* picks the binding radius for the first forward reaction that is listed. Thus, if the product parameter for dissociation of X is p , the requested geminate rebinding probability will be found for the reaction $A+B \rightleftharpoons C$, but a different value will be found for the reaction $A+B \rightleftharpoons D$.



This reaction scheme might represent two different pathways by which A and B can bind to form an identical complex. However, *Smoldyn* cannot tell which reverse reaction corresponds to which forwards reaction. Instead, for both determining the binding and unbinding radii, it uses the first reverse reaction that is listed.

The general principle for calculating binding radii is that *Smoldyn* first looks to see if a reaction is directly reversible (*i.e.* as in example 3, without any consideration of reaction network loops or other possible causes of geminate reactions). If it is and if the reversible parameter is p , x , r , or b , then the binding radius is found under the assumption that the rate constant was measured using just this reaction, at equilibrium. If not, or if the reversible parameter is q , y , s , o , f , or i , then *Smoldyn* calculates the binding radius with the assumption that the rate constant was measured using just that reaction at steady-state and with all product removed as it is formed.

Unbinding radii typically require a reversible parameter (except as in example 2). If the parameter is b , o , or f , the requested unbinding radius is used. If it is i , the unbinding radius is set to 0. Otherwise, it can only be calculated with the knowledge of the binding radius. If the reaction is directly reversible, the binding radius for the reverse reaction is used. If it is not directly reversible but the products can react, as in examples 5, 6, and 8, then the binding radius for the first reaction that is listed is used.

3.2 Time steps

The simulated time in *Smoldyn* increases with discrete increments. However, a major focus of program design has been to make it so that results are indistinguishable from those that would be obtained if the simulated time increased continuously. This goal cannot be achieved perfectly. Instead, the algorithms are written so that the simulation approaches the Smoluchowski description of reaction-diffusion systems as the time step is reduced towards zero, and so it maintains as much accuracy as possible for longer time steps. This topic is discussed in detail in the research paper “Stochastic

simulation of chemical reactions with spatial resolution and single molecule detail” by myself and Dennis Bray (it was submitted to *J. Phys. Chem.* in September 2002). Some more discussion of this topic is given here.

In concept, the system is observed at a fixed time, then it evolves to some new state, then it is observed again, and so forth. A complication is that commands allow the user to manipulate the system at fixed times; it is typically best for these manipulations to immediately precede observations. For example, if a command states that some collection of molecules should be removed at time t , then an observation that is also at time t should show that they have been removed. This leads to the following sequence of program operations:

```

----- time = t -----
  manipulate system
  observe system
  diffuse molecules
  surface interactions
  reactions
    0th order reactions
    1st order reactions
    2nd order reactions
    add reaction products to system
  surface interactions
----- time = t +  $\Delta t$  -----

```

To follow this scheme, manipulation commands should be entered before observation commands (the other order is possible as well if observations are desired before manipulations). After commands are run, graphics are displayed to OpenGL if that is enabled. The evolution over a finite time step starts by diffusing all mobile molecules. In the process, some end up across the walls of the boundary and others are within the binding radii of other reactants. Wall collisions are treated by reflecting molecules back into the simulation volume (for reflective boundaries). Next, reactions are treated in a semi-synchronous fashion. They are asynchronous in that all zeroth order reactions are simulated first, then unimolecular reactions, and finally bimolecular reactions. With bimolecular reactions, if a molecule is within the binding radii of two different other molecules, then it ends up reacting with only the first one that is checked, which is arbitrary. Reactions are synchronous in that reactants are removed from the system as soon as they react and products are not added into the system until all reactions have been completed. This prevents reactants from reacting twice during a time step and it prevents products from one reaction from reacting again during the same time step. As it is possible for reactions to produce molecules that are outside the system walls, those products are then reflected back into the system. At this point, the system has fully evolved by one time step. All molecules are inside the system walls and essentially no pairs of molecules are within their binding radii (the exception is that products of a bimolecular reaction with an unbinding radius might be initially placed within the binding radius of another reactant).

Each of the individual routines that is executed during a time step exactly produces the results of the Smoluchowski description, or yields kinetics that exactly match those that were requested by the user. However, the simulation is not exact for all length time steps because it cannot exactly account for interactions between the various phenomena. For example, if a system was simulated that only had unimolecular reactions and the products of those reactions did not react, then the simulation would yield exactly correct results using any length time step. However, if the products could react, then there are interactions between reactions and there would be errors. In this case, the error arises because *Smoldyn* does not allow a molecule to be in existence for less than the length of one time step.

3.3 Reactions near surfaces

Smoldyn does not treat reactions near walls any differently from other reactions. Moreover, if walls are reflective and a reactant happens to be less than a binding radius from a wall, then part of the “binding volume” is inaccessible to other potential reactants; again, there is no special treatment for this.

When molecules have excluded volume, even inert impermeable surfaces can affect the local concentrations of chemicals. The obvious effect is that a molecule cannot be closer to a surface than its radius, leading to a concentration of zero closer than that. In a mixture of large and small molecules, Brownian motion tends to push the large molecules up against surfaces while the small molecules occupy the center of the accessible volume, creating more complex concentration effects. These effects do not occur when excluded volume is ignored, as it is in *Smoldyn*, in which case surfaces do not affect local concentrations.

While surfaces do not affect concentrations of non-reacting molecules, they do affect reaction rates. Consider the reaction $A+B \rightarrow C$, where A is fixed and B diffuses. If essentially all A molecules are far from a surface, the diffusion limited reaction rate is found by solving the diffusion equation for the radial diffusion function (RDF) with the boundary conditions that the RDF approaches 1 for large distances and is 0 at the binding radius (see the paper by myself and Dennis Bray titled “Stochastic simulation of chemical reactions with spatial resolution and single molecule detail”). This leads to the Smoluchowski rate equation

$$k = 4\pi D b$$

However, for an A molecule that is near a surface, an additional boundary condition is that the gradient of the 3 dimensional RDF in a direction perpendicular to the surface is zero at the surface. This makes the solution of the reaction rate sufficiently difficult that I have not attempted to solve it, but the result is different from the simple result given above. This surface effect is an issue whenever the A molecule is within several binding radii of a surface and is especially pronounced when it is closer to the surface than its binding radius. For cases in which the A molecule is more than one binding radius from the surface, B molecules are going to take longer than usual to reach the region between the A and the surface, leading to a decreased reaction rate. It is suspected that the

reaction rate decreases monotonically as the A molecule approaches and then crosses a surface.

A special case that can be solved exactly occurs when the A molecule is exactly at the surface, such that half of the binding volume is accessible to B molecules and half is inaccessible. Now, the RDF inside the system volume is identical to the RDF for the case when the A molecule is far from a surface. The logic is to assume that this is true and to then observe that it already satisfies the additional boundary condition. Using this RDF, the diffusive flux is half of the diffusive flux for an A molecule far from a surface, because only half of the binding surface is exposed to the system. Thus, the diffusion limited reaction rate for the situation in which a reactant is fixed exactly at a surface is

$$k = 2\pi D b$$

The situation changes some when simulation time steps are sufficiently long that rms step lengths are much longer than binding radii. Now, the probability of a reaction occurring during a time step is a function of only the binding volume. Thus, there are no surface effects at all when an A molecule is fixed anywhere in the simulation volume that is greater than or equal to one binding radius away from a surface. As the A molecule is moved closer to the surface, the reaction rate decreases in direct proportion to the binding volume that is made inaccessible to B molecules. An especially easy situation is that when the A molecule is exactly at the surface, the reaction rate is half of its value when the A molecule is far from a surface, which is the same as the diffusion limited result.

In conclusion, reaction rates are reduced near surfaces and the effect is different for diffusion limited and activation limited reactions. However, for both cases, and almost certainly for all cases in between, the reaction rate is exactly half when an A molecule is fixed at a surface, compared to when it is far from a surface. A few tests with *Smoldyn* using the files *reactW#*, described below, suggested that these surface effects are likely to be minimal for most situations, although it is good to be aware of their potential. The exception is that there are large surface effects when molecules are fixed with a significant portion of the binding volume outside the simulation volume.

4. Tests of *Smoldyn*

4.1 Initial test: *bounce1*, *bounce2*, and *bounce3*

The simplest test is just to make sure that the *Smoldyn* application is able to launch and run properly, using a very simple configuration file. These tests were also useful for getting the graphical output to work properly. Each file just shows a collection of molecules that bounce around inside the system walls. *bounce1* works in one dimension, *bounce2* in two dimensions, and *bounce3* in three dimensions. When running *bounce3*, note that the arrow keys can be used to rotate the system about the middle, the '=' and '-' keys zoom in and out, the 'x', 'X', 'y', 'Y', 'z', and 'Z' keys rotate the system about the individual axes (sometimes rotating the object out of the visible region), the space bar pauses the simulation, the '0' key resets rotations and zooming to initial values, and 'Q' quits the simulation.

4.2 Diffusion rates: *diffi* and *diffa*

diffi was used to quantitatively test isotropic diffusion by letting several collections of molecules diffuse outwards from the center of space. The moments of the molecular distributions are saved as functions of time. The zeroth moment is the number of molecules, which obviously ought to stay at a constant value with no fluctuations. This was verified. The first moment is a vector quantity representing the mean position of the set of molecules. Because of symmetry, its value should stay near the initial position (the origin), although fluctuations are expected. These fluctuations are given with the equation

$$|\text{mean} - \text{starting point}| \propto \sqrt{\frac{2Dt}{n}}$$

D is the diffusion coefficient, t is the simulation time, and n is the number of molecules. This equation agrees well with simulation data. The second moment is a matrix quantity which gives the variance on each pair of axes of the distribution of positions. For example, the variance matrix element for axes x and y is

$$v_{xy} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

The overbars indicate mean values for the distribution. Equations are analogous for other pairs of axes. Because diffusion on different axes is independent, the off-diagonal variances (v_{xy} , v_{xz} , and v_{yz}) are expected to be about 0, but with some fluctuations, which was verified. However, the diagonal variances (v_{xx} , v_{yy} , and v_{zz}) are each expected to increase as approximately

$$v_{xx} \propto v_{yy} \propto v_{zz} \propto 2Dt$$

This behavior was verified for the simulation data, using a variety of simulation time steps and diffusion coefficients. However, the fluctuations of the variances were not analyzed.

diffa was used to investigate anisotropic diffusion. In this case, the diffusion equation is

$$\dot{u} = \nabla \cdot \mathbf{D} \nabla u$$

u can be interpreted as either the probability density for a single molecule or as the concentration of a macroscopic collection of molecules. \mathbf{D} is the physical diffusion matrix, which is the square of the matrix that is entered in the configuration file (matrix square roots can be calculated with MatLab, Mathematica, or other methods). If \mathbf{D} is equal to the identity matrix times a constant, D , the equation reduces to the standard

isotropic diffusion equation. Using the file *diffa*, it was verified that the mean position for anisotropic diffusion is the initial position using diffusion matrices with and without off-diagonal elements. Using molecules that diffused in just the x - z plane, the variance on each axis was confirmed to be nearly equal to $2Dt$, using the D value for each axis. It was also verified that the more complicated diffusion matrix yielded qualitatively reasonable diffusion, although the variances were not checked quantitatively.

4.3 Zeroth order reaction rates: *zeroreactf*, *zeroreactm*, *zeroreacts*

The zeroth order reaction nothing \rightarrow A proceeds with the mass action rate equation

$$n(t) = n(0) + kt$$

$n(0)$ and $n(t)$ are the initial and time dependent numbers of A molecules and k is the zeroth order reaction rate. Using the file *zeroreactf*, *zeroreactm*, and *zeroreacts* (fast, medium, and slow) it was confirmed that the simulation results conform closely to the theoretical result, using rates ranging from 100 molecules per simulation time step to 0.01 molecules per time step. The simulation showed fluctuations in the production rates, as expected, but these were not analyzed.

4.4 Unimolecular reaction rates: *unireact1*, *unireactn*

unireact1 was used to check unimolecular reaction rates using a wide range of reaction rates, and thus a wide range of reaction probabilities in each time step. Each molecular species defined in the configuration file has a single unimolecular reaction pathway and simply goes away when it reacts. These chemical equations are each equivalent to simply $A \rightarrow \text{nothing}$. The theoretical rate equation is

$$n(t) = n(0)e^{-kt}$$

$n(t)$ is the number of molecules remaining after time t , $n(0)$ is the initial number of molecules, and k is the first order rate constant. It was confirmed that simulation data agreed well with the equation using reaction probabilities that ranged from 0.001 to 0.632 per time step.

unireactn tests reaction probabilities using multiple reaction mechanisms. For convenience in analysis, the reactant concentration is kept constant by having the reactant as a reaction product. The reactions are



The system is started with only A molecules, so the theoretical number of B molecules as a function of time is

$$n_B(t) = k_B n_A t$$

Analogous equations hold for C and D. Simulation results closely matched these theoretical equations when a small time step was used. However, a large time step leads to a relatively large probability that an individual A molecule will react by one of the three pathways during the time step. If this probability is p (the sum of the probabilities for each of the three reaction pathways), then the probability that an A should react twice during the same time step is proportional to p^2 . In the simulation, a molecule can only react once during a time step, leading to errors whenever p^2 is large, and hence when p is large.

4.5 Bimolecular reactions, different reactants: *reactABs*, *reactABm*, *reactABf*

The reaction $A+B \rightarrow C$ is easy to analyze using mass action kinetics with the condition that there are the same numbers of A and B molecules initially. The solution for the number of A molecules (or B molecules) as a function of time is

$$n(t) = \frac{1}{n(0)} + \frac{kt}{V}$$

reactABs, *reactABm*, and *reactABf* (slow, medium, and fast) were used to test the simulated reaction rate. In all cases, but especially with the fast reaction rate, the simulated rate is faster initially than the analytical rate because the simulation starts with molecules randomly distributed whereas the analytical result assumes a steady-state distribution. However, after enough time passes for a steady state distribution to be formed, the simulated results agree quite well with the analytical results. These files examine reaction rates that have ratios of mutual rms step lengths to binding radii ranging from 0.146 (*bireactABf*, diffusion limited) to 2.15 (*bireactABs*, activation limited).

4.6 Bimolecular reactions, same reactant: *reactAAs*, *reactAAm*, and *reactAAf*

The reaction $A+A \rightarrow C$ was investigated as well. Using mass action kinetics, the analytical solution for the number of A molecules as a function of time is

$$n(t) = \frac{1}{n(0)} + \frac{2kt}{V}$$

This was tested using the files *reactAAs*, *reactAAm*, and *reactAAf*. All files agreed well with the analytical equation. The configuration file ratios of mutual rms step length to binding radius ranged from 0.145 for the fast version (diffusion limited) to 1.67 for the slow version (activation limited). As before, the simulated reaction rate was faster

initially than the analytical rate because of different initial molecule distributions. However, the rates agreed well after the simulation distribution had time to reach a steady-state profile.

4.7 Reactions near walls: *reactW1*, *reactW2*, *reactW3*, *reactW4*

In principle, an impermeable surface should not change the local concentration of a reactant, although it can affect reaction rates. This was investigated using the same system as for the $A+B \rightleftharpoons C$ reaction shown above, except that all A molecules were fixed near the surfaces of the simulation volume. The binding radius for all files is 0.763 and the rms step length of the B molecules is 0.632, leading to reactions that are intermediate between diffusion and activation limited. In the file *reactW1*, the A molecules are positioned 5 units inside reflective walls, which is large compared to the rms step length or binding radius. Kinetics are quite different from those described in section 4.5 because of the correlated A molecule positions and surface effects, although an analytical equation was not derived. Changing the distance from the walls to 1 unit in *reactW2*, which is now close to the rms step length and binding radius, makes essentially no difference in the output. In *reactW4*, the A molecules are placed on the walls and the boundaries are made periodic, which again has no effect. Other tests involved increasing or decreasing the time steps by a factor of 10 for these files, which also had no significant effects. Thus, surface effects are minimal for reaction rates when the entire binding volume is accessible to diffusing molecules.

In *reactW3*, the A molecules are placed on the walls and the boundaries are reflective so that only half of the binding volume is accessible. The reaction rate in this case should be exactly half of the rate for the other files, although it was consistently found that the ratio of the rates is closer to 0.55, for a wide range of time step lengths. The cause of this difference has not been identified.

4.8 Reversible reactions: *equil*

Reversible reactions involve geminate recombination issues, as discussed in section 3. The accuracy of reversible reaction rates was investigated with the configuration file *equil*, in which an equilibrium is set up for the reaction $A+B \rightleftharpoons C$. From standard chemistry, the equilibrium constant is related to the ratio of product to reactant concentrations and to the ratio of the forward to reverse rate constants,

$$K = \frac{n_C V}{n_A n_B} = \frac{k_f}{k_r}$$

V is the total system volume. The configuration file *equil* starts with equal numbers of A and B molecules and no C molecules. Using the above equation and this starting point, the solution for the equilibrium number of A molecules is

$$n_A = \frac{-V + \sqrt{V^2 + 4Kn_A(0)V}}{2K}$$

$n_A(0)$ is the initial number of A molecules. It was verified that the simulation result approached this value. As usual, fluctuations were not analyzed.

4.9 Simple reaction networks: *lotvolt*

The file *lotvolt* just runs a simple Lotka-Volterra system of reactions to make sure that the various components of *Smoldyn* work together. This configuration file is identical to the one shown in section 2.5. Results were not analyzed quantitatively.

5. Copyright and Citation

Nearly all of the components of *Smoldyn* were written by myself (Steven Andrews), with the exceptions being a few short routines that were copied from *Numerical Recipes in C* (Press, Flannery, Teukolsky, and Vetterling, Cambridge University Press, 1988), which are acknowledged where appropriate. The compiled version of *Smoldyn*, the components of the source code that are not copyrighted by others, and this documentation are copyrighted by myself. However, permission is granted for any non-commercial use of the program and of the source code. The only portion of the code that may not be modified is the copyright information. No warranty is made for the performance or suitability of any portion of *Smoldyn*.

I expect to maintain a working copy of the program indefinitely. The current download site for *Smoldyn* is <http://sahara.lbl.gov/~sandrews/index.html>, where the program may be obtained for free. If improvements are made to the code or bugs are fixed, then I would appreciate a copy of the modified source code. If you find any bugs in the code, please let me know! My e-mail address is ssandrews@lbl.gov.

If *Smoldyn* is used to a significant extent, it may be appropriate to cite or acknowledge its use.

Smoldyn Code Documentation

As a program without documented code is both unwritable and unmaintainable, here is the documentation. As can be seen from looking at the code, I don't particularly like comments in the code itself or excessive white space. However, the description below should be complete for the main source files. Other files are documented in my library documentation.

The main code is separated into three files. `smoldyn.c` contains the `main()` routine and some high level routines for running the simulation. It also includes essentially all of the OpenGL graphics routines. `smollib.c` and its header `smollib.h` contain all structure declarations and all low level routines, including ones that allocate and free memory, calculate simulation parameters, run the simulation, and provide diagnostics. This is the core of the program and, hopefully, should rarely require modifications. `smollib2.c` and its header `smollib2.h` include all runtime interpreter commands. It is expected that more commands will be desired on a regular basis, so it is expected that this library will be appended regularly. As the file `smollib.c` is written in ANSI C and does not contain any graphics calls, it should be completely portable. The other files should be portable to any system that supports OpenGL, and can be modified trivially to run on other systems.

1. Include files

smoldyn.c, non-OpenGL

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "smollib.h"
#include "smollib2.h"
#include "string2.h"
#include "SimCommand.h"
```

smoldyn.c, with OpenGL

```
#include "opengl2.h"
#include "gl.h"
#include "glut.h"
```

smollib.h

```
#include "SimCommand.h"
```

smollib.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "Rn.h"
#include "Zn.h"
#include "random.h"
#include "string2.h"
#include "math2.h"
```

```
#include "rxnparam.h"
#include "SimCommand.h"
#include "smolib.h"
#include "smolib2.h"
#include "VoidComp.h"
```

smolib2.h

```
#include "SimCommand.h"
#include "smolib.h"
```

smolib2.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "Rn.h"
#include "Zn.h"
#include "random.h"
#include "string2.h"
#include "smolib2.h"
#include "opengl2.h"
#include "SimCommand.h"
```

These are simple and self-explanatory.

2. Constants, macros, and global variables

smoldyn.c

```
#define SMOLDYN_VERSION 1.53
simptr Sim;
int Vb,*Ctr;
time_t Tstt;
```

smolib.c

```
#define RANDTABLEMAX 4095
#define CHECK(A) if(!(A)) goto failure
#define CHECKS(A,B) if(!(A)) {strncpy(erstr,B,STRCHAR);goto failure;}
float GaussTable[RANDTABLEMAX+1];
```

The notation used is that macros and constants defined with the pre-processor are in all capitals, global variables are preceded by a capital letter, and local variables are in all lower case. Variables that have a meaning have meaningful names, whereas those that are scratch space have generic names that simply indicate the variable types. All of these macros and global variables are global only within their source file, so that they are not accessible to other files.

SMOLDYN_VERSION is the current version number of Smoldyn. This number has been fluctuating greatly, although with an upward trend, but I'll try to make it more useful in the future.

`Sim` is a global variable for the current simulation structure. This, as well as the other global variables in `smoldyn.c` are only used when graphics are being shown using OpenGL, because OpenGL does not allow variables to be passed in the normal way between functions.

`Vb` is a global flag for verbose operation, equal to 1 if yes, 0 if no.

`Ctr` is a global array of simulation counters, which count the total number of events that occur during a simulation. Index 0 is for zeroth order reactions, 1 is for first order reactions, 2 is for second order reactions within a partition, 3 is for second order reactions between partitions, and 4 is for wall collisions.

`Tstt` is a global variable for the starting time of simulation execution, allowing the total runtime to be determined.

`RANDTABLEMAX` is the maximum element number of the random number conversion table, which is used both for allocating the table (with 1 more element) and as a bit mask for random number routines. Note that it needs to be 1 less than an integer power of 2.

`CHECK` is a useful macro for several routines in which any of several problems may occur, but all problems result in freeing structures and leaving. Program flow goes to the label `failure` if `A` is false. Many people would consider both the use of a macro function and the use of a `goto` statement to be bad programming practice, and especially bad when used together. However, in this case it significantly improves code readability. As usual, partially defined structures should always be kept traversable and in good order so they can be freed at any time. However, there is one subtle situation where `CHECK` can cause surprising behavior, which must be looked out for:

```
if(test)
    CHECK(a==b);
else {... }
```

WRONG

The problem is that `CHECK` is a macro for an `if()` statement, so the `else` in the above example becomes an `else` for the `CHECK`, rather than an `else` for the `if(test)` portion, as intended. Instead, this should be coded with braces:

```
if(test) {
    CHECK(a==b); }
else {... }
```

RIGHT

`CHECKS` is identical to `CHECK`, except that it also copies the included string to the variable `erstr` if a failure occurs. This is useful for error reporting. The same comments made above are important here as well.

`GaussTable` is a table to convert uniform random values to normally distributed ones.

3. Local variables

It has proven useful to use consistent names for local variables for code readability. In places, there are exceptions, but the following table lists the typical uses for most local variables:

| variable | type | use |
|-------------|---------------|--|
| a | float | binding radius for bimolecular reaction |
| b,b2 | int | box address |
| blist | boxptr* | list of boxes, index is [b] |
| boxs | boxssptr | pointer to box superstructure |
| bptr | boxptr | pointer to box |
| bval | float | unbinding radius for bimolecular reaction |
| ch | char | generic character |
| cmd | cmdptr | pointer to a command |
| cmds | cmdssptr | pointer to the command superstructure |
| d | int | dimension number |
| dc1,dc2 | float | diffusion coefficients for molecules |
| dead | moleculeptr* | list of dead molecules, index [m] |
| difc | float* | list of diffusion coefficients, index [i] |
| dsum | float | sum of diffusion coefficients |
| dim | int | dimensionality of space |
| dt | float | time step |
| er | int | error code |
| erstr | char* | error string |
| flt1,flt2 | float | generic float variable |
| fptr | FILE* | file stream |
| got | int[] | flag for if parameter is known yet |
| i | int | molecule identity, reactant number, or generic integer |
| i1,i2,... | int | molecule identities |
| indx | int* | dim dimensional index of box position |
| itct | int | count of number of items read from a string |
| j | int | number of reaction for certain i |
| lctr | int | line number counter for reading text file |
| line | char[] | complete line of text |
| line2 | char* | pointer to unparsed portion of string |
| live | moleculeptr** | list of live molecules, index [l1][m] |
| ll | int | index of live list |
| m,m2,m3 | int | index of molecule in list |
| m1,m2,m3 | int* | scratch space matrices of size dimxdim |
| mlist | moleculeptr* | list of molecules, index is [m] |
| mols | molssptr | pointer to molecule superstructure |
| mptr | moleculeptr | pointer to molecule |
| mptr1,mptr2 | moleculeptr | pointer to more molecules |

| | | |
|----------|---------------|---|
| name | char** | names of molecules, index is [i] |
| nbox | int | number of boxes |
| ni2o | int | value of $nident^{order}$ |
| nident | int | number of molecule identities |
| nl | int* | number of live molecules in a live list, index [11] |
| nm,nm2 | char[] | name of molecule or reaction |
| nmol | int | number of molecules in list |
| nprod | int*,int | number of products for reaction [r] |
| nrxn | int* | number of reactions for [i] |
| o2 | int | order of second reaction |
| optr | int* | pointer to the order of reaction |
| order | int | order of reaction |
| p | int | reaction product number |
| pgeomptr | float* | pointer to probability of geminate recombination |
| prod | moleculeptr** | list of products for reaction [r], index is [p] |
| r | int | reaction number |
| r2 | int | reaction number for second reaction |
| rate | float* | requested rate of reaction [r] |
| rate2 | float* | internal rate parameter of reaction [r] |
| rate3 | float | actual rate of reaction |
| rev | int | code for reaction reversibility; see findreverserxn |
| rname | char** | names of reactions, index is [r] |
| rpar | float,float* | reversible parameter, index is [r] |
| rpart | char,char* | reversible parameter type, index is [r] |
| rptra | int* | pointer to reaction number |
| rxn | rxnptr | pointer to a reaction structure |
| rxn2 | rxnptr | pointer to a second reaction structure |
| side | int* | number of boxes on each side of space, index [d] |
| sim | simptr | pointer to simulation structure |
| smpttr | simptr* | pointer to pointer to simulation structure |
| step | float | rms step length of molecule or molecules |
| str1 | char[] | generic string |
| table | int** | table of reaction numbers for [i][j] |
| top | int | top of empty molecules in dead list |
| total | int | total number of reactions in list |
| v1,v2,v3 | int* | scratch space vectors of size dim |
| w | int | index of wall |
| wlist | wallptr* | list of walls, index is [w] |
| word | char[] | first word of a line of text |
| wptr | wallptr | pointer to wall |

4. Structures, allocation, and freeing routines – smolib.h, smolib.c

While *Smoldyn* is written in C, it uses an object oriented approach to programming, making the proper maintenance of structures one of the central aspects of the program.

The structures are described below. In general, the basic objects are molecules, walls, and virtual boxes, each of which has its own structure. In many cases, these items are grouped together into superstructures, which are basically just a list of fundamental elements, along with some more information that pertains to the whole list. Finally, a simulation structure is a high level structure which contains all the parameters and the current state of the simulation.

An aspect of structures that is important to note, especially if changes are made, is which structures own what elements. For example, a molecule owns its position vector, meaning that that piece of memory was allocated with the molecule and will be freed with the molecule. On the other hand, a molecule does not own a virtual box, but merely points to the one that it is in.

All allocation routines return either a pointer to the structure that was allocated, or NULL if memory wasn't available. Assuming that they succeed, all structure members are initialized, typically to 0 or NULL depending on the member type. All the memory freeing routines are robust in that they don't mind NULL inputs or NULL internal pointers. However, this is only useful and robust if allocation is done in an order that always keeps the structure traversable and keeps pointers set to NULL until they are ready to be initialized.

Molecules

```
typedef struct moleculestruct {
    float *pos;                dim dimensional vector for position
    float *posx;               dim dimensional vector for old position
    int ident;                 identity of molecule; 0 is empty
    struct boxstruct *box; } *moleculeptr; pointer to box which molecule is in
```

moleculestruct (declared in smollib.h) is a structure used for each molecule. pos and posx, both of which are owned by the structure, are always valid positions, although not necessarily within the system volume. posx is the position from the previous time step, used to determine if a molecule crossed a surface. ident should always be between 0 and nident-1, inclusive. A molecule type of 0 is an empty molecule for transfer to the dead list. Except during setup, box should always point to a valid box.

```
moleculeptr molalloc(int dim)
    molalloc allocates and initializes a new moleculestruct. The box and diffusion
    matrix members are returned as NULL.
```

```
void molfree(moleculeptr mptr)
    molfree frees the space allocated for a moleculestruct, as well as its two position
    vectors.
```

See also:

assignmolecs - assigns molecules to boxes, setting the box pointer

Molecule superstructure

| | |
|--|--|
| <code>typedef struct molsuperstruct {</code> | |
| <code>float *difc;</code> | diffusion constants for each identity |
| <code>float *difstep;</code> | rms diffusion step for each identity |
| <code>float **difm;</code> | diffusion matrix for each identity |
| <code>int *display;</code> | size of molecule in graphical display |
| <code>float **color;</code> | RGB color vector for each identity |
| <code>moleculeptr *live[2];</code> | live molecules in system (0 mobile, 1 fixed) |
| <code>moleculeptr *dead;</code> | list of dead molecules |
| <code>int max;</code> | size of each molecule list |
| <code>int nl[2];</code> | number of molecules in live lists |
| <code>int nd;</code> | total number of molecules in dead list |
| <code>int top; } *molssptr;</code> | index for dead list; above are resurrected |

`molsuperstruct` (declared in `smollib.h`) contains and owns information about molecular properties and it also contains and owns three lists of molecules. Diffusion is described with `difc`, which is a `nident` length vector of diffusion constants; `difstep` is a `nident` length vector of the rms displacements on each coordinate during one time step if diffusion is isotropic; and `difm` is a `nident` length list where each element is either a `NULL` value if diffusion is isotropic or a `dimxdim` size diffusion matrix (actually the square root of the matrix). For the molecule lists, the former is separated into two parts. The first set is the `live` list, which are those molecules that are actually in the system; the others are in the `dead` list, are empty molecules, and have no influence on the system. The two parts of the `live` list are the mobile molecules (`live[0]`) and the fixed molecules (`live[1]`). They are differentiated solely by whether their diffusion constants are zero and are separated into two lists to speed up bimolecular reaction routines. All lists have size `max`. Upon initialization, all molecules are created as empty molecules in the `dead` list and both `live` lists full of `NULL`s, whereas during program execution, all lists are typically partially full. When lists are properly ordered, each `live` list, `ll`, has molecules from element 0 to element `nl[ll]-1`, inclusive, and is filled with `NULL`s from `nl[ll]` to `max-1`. Similarly, the `dead` list is filled with empty molecules from 0 to `nd-1`, and `NULL`s from `nd` to `max-1`; in this case, `top` is equal to `nd`. Chemical reactions destroy molecules, turning `live` ones into empty molecules, and they create new molecules, in which `dead` ones are resurrected. After a reaction, there are generally some empty molecules scattered throughout the `live` lists and some active molecules in the `dead` list, from `top` to `nd-1`, inclusive. While these mis-sorted molecules should not cause a problem with other routines, they should be sorted relatively promptly. No order is maintained for the molecules within the respective `live` lists. The total number of molecules in all the lists is constant and is equal to `max`. If more molecules are needed in the system than the total number allocated, the program sends an error message and ends; in the future, it may be possible to dynamically create larger lists.

Example of the lists:

| index | live[0] | | live[1] | | dead | |
|-------|---------|---|---------|---|------|---|
| 8 | max | ? | max | ? | max | ? |
| 7 | | - | | - | | - |
| 6 | | - | | - | | - |
| 5 | | - | | - | | - |
| 4 | | - | | - | | - |
| 3 | nl[0] | - | | - | nd | - |
| 2 | | 2 | nl[1] | - | top | 1 |
| 1 | | 1 | | 0 | | 0 |
| 0 | | 0 | | 3 | | 0 |

Here, each list has $max=8$, and so is indexed with m from 0 to 7. A '?' is memory that is not part of that which was allocated, a '-' is a NULL value, a '0' is an empty molecule, and other numbers are other identities ('1' and '2' are mobile, whereas '3' is immobile). The '0's in the the two live lists are to be transferred to the dead list during the next sort, while the '1' in the dead list is to be moved to mobile live list.

`molssptr molssalloc(int dim,int max,int nident);`
 molssalloc allocates and initiallizes a molecule superstructure with max molecule spaces in each of the three lists. max must be at least 1. The dead list is filled with empty molecules. The molecule boxes are left as NULLs, and need to be set. Book keeping elements for the lists are set to their initial values.

`void molssfree(molssptr mols,int nident);`
 molssfree frees both a superstructure of molecules and all the molecules in all its lists.

See also:

`molssort` - sorts the live and dead molecules, putting them in the proper lists.

`molssoutput` - displays and checks many parameters of a molecule superstructure.

Walls

```
typedef struct wallstruct {
    int wdim;           dimension number of perpendicular to wall
    int side;           low side of space (0) or high side (1)
    float pos;          position of wall along dim axis
    char type;          properties of wall
    struct wallstruct *opp; } *wallptr;  pointer to opposite wall
```

wallstruct (declared in `smollib.h`) is a structure used for each wall. The type may be one of four characters, representing the four possible boundary conditions.

| <u>type</u> | <u>boundary</u> |
|-------------|-----------------|
| r | reflecting |
| p | periodic |
| a | absorbing |
| t | transparent |

Pointers to the opposite walls are used for wrap-around diffusion, but are simply references. There is no superstructure of walls, but, instead a list of walls is used. Walls need to be in a particular order: walls numbered 0 and 1 are the low and high position walls for the 0 coordinate, the next pair are for the 1 coordinate, and so on up to the $2 \cdot \text{dim} - 1$ wall. These walls are designed to be bounds of simulated space, and are not configured well to act as membranes.

```
wallptr wallalloc(void);
```

wallalloc allocates and initializes a new wall. The pointer to the opposite wall needs to be set.

```
void wallfree(wallptr wptr);
```

wallfree frees a wall.

```
wallptr *wallsalloc(int dim);
```

wallsalloc allocates an array of pointers to $2 \cdot \text{dim}$ walls, allocates each of the walls, and sets them to default conditions (reflecting walls at 0 and 1 on each coordinate) with correct pointers in each opp member.

```
void wallsfree(wallptr *wlist, int dim);
```

wallsfree frees an array of $2 \cdot \text{dim}$ walls, including the walls.

See also:

checkwalls - takes care of proper molecule behavior at walls.

walloutput - displays most parameters of a collection of walls.

Boxes

```
typedef struct boxstruct {
```

```
    int *indx;
```

dim dimensional index of the box

```
    int nneigh;
```

number of neighbors in list

```
    int midneigh;
```

logical middle of neighbor list

```
    struct boxstruct **neigh;
```

all box neighbors, using sim. accuracy

```
    int *wpneigh;
```

wrapping code of neighbors in list

```
    int nwall;
```

number of walls in box

```
    wallptr *wlist;
```

list of walls that cross the box

```
    int maxmol[2];
```

allocated size of live lists

```
    int nmol[2];
```

number of molecules in live lists

```
    moleculeptr *mol[2]; } *boxptr;
```

lists of live molecules in the box

boxstruct (declared in `smollib.h`) is a structure for each of the virtual boxes that partition space. Each box has a list of its neighbors, in `neigh`, as well as a little information about them. This list extends from 0 to `nneigh-1`. From 0 to `midneigh-1` are those neighbors that logically precede the box, meaning that they are above or to the left, whereas those from `midneigh` to `nneigh-1` logically follow the box. If there are no periodic boundary conditions, the logical order is the same as the address order; however, this is not necessarily true with the inclusion of wrap-around effects. In `wpneigh` is a code for each neighbor that describes in what way it is a neighbor: 0 means that it's a normal neighbor with no edge wrap-around; otherwise pairs of bits are associated with each dimension (low order bits for low dimension), with the bits equal to 00 for no wrapping in that dimension, 01 for wrapping towards the low side, and 10 for wrapping towards the high side. This might be clearer in the Zn.c documentation. The neighbors that are listed depend on the requested simulation accuracy:

| <u>accuracy</u> | <u>neighbors</u> | <u>wrap-around</u> |
|-----------------|------------------|--------------------|
| <3 | none | no |
| 3 to <6 | nearest | no |
| 6 to <9 | nearest | yes |
| >9 | all | yes |

Boxes also have lists of mobile molecules (`mol[0]`, allotted to size `maxmol[0]`, and filled from 0 to `nmol[0]-1`), immobile molecules (`mol[1]`, etc.) molecules, and walls (`wlist`, allocated and filled with `nwall` pointers) within them. While the lists are owned by the box, the members of the lists are simply references, rather than implications of ownership. The same, of course, is true of the neighbor list, although the box owns the `wpneigh` list. If wall or neighbor lists are empty, the list is left as NULL, whereas the molecule list always has a few spaces in it.

```
boxptr boxalloc(int dim);
```

`boxalloc` allocates and minimally initializes a new `boxstruct`. The only list allocated is `indx`, which is set to 0's.

```
void boxfree(boxptr bptr);
```

`boxfree` frees the box and its lists, although not the structures pointed to by the lists.

```
boxptr *boxesalloc(int dim,int nbox);
```

`boxesalloc` allocates and initializes an array of `n` boxes, including the boxes. Again, initialization is minimal, with only the `indx` array of the boxes allocated, which is set to 0's.

```
void boxesfree(boxptr *blist,int nbox);
```

`boxesfree` frees an array of boxes, including the boxes.

See also:

`expandbox` - expands the size of the molecule lists of a box.

`boxoutput` - displays most parameters of a box.

Box superstructure

| | |
|--|---|
| <code>typedef struct boxsuperstruct {</code> | |
| <code>float mpbox;</code> | requested number of molecules per box |
| <code>float boxsize;</code> | requested box width |
| <code>int nbox;</code> | total number of boxes |
| <code>int *side;</code> | number of boxes on each side of space |
| <code>float *min;</code> | position vector for low corner of space |
| <code>float *size;</code> | length of each side of a box |
| <code>boxptr *blist; } *boxssptr;</code> | actual array of boxes |

`boxsuperstruct` (declared in `smollib.h`) expresses the arrangement of virtual boxes in space, and owns the list of those boxes and the boxes. Either `mpbox` or `boxsize` are used and not both. Boxes are arranged in a rectanguloid grid and exactly cover all space inside the walls. The structure of the boxes in space is the same as that of a `dim` rank tensor, allowing tensor indexing routines to be used to convert between box addresses and indicies. The box index along the d 'th dimension of a point with position `x[d]` is

$$\text{indx}[d] = (\text{x}[d] - \text{min}[d]) / \text{size}[d];$$

where integer arithmetic takes care of the truncation. Converging from box index to address is easy with the tensor routine in `Zn.c`, or can also be calculated quickly with the following code fragment, which outputs the box number as `b`,

```
for(b=0,d=0;d<dim;d++)    b=side[d]*b+indx[d];
```

Converting the box number to the indicies can also be done, but the `Zn.c` routine is easiest for this.

`boxssptr boxssalloc(int dim);`
`boxssalloc` allocates and initializes a superstructure of boxes, including arrays for the `side`, `min`, and `size` members, although the boxes are not added to the structure; *i.e.* `blist` is set to `NULL` and `nbox` is 0. Initial values for `side` and `size` members are all set to 1, `min` values are set to 0, and `mpbox` is set to 5; all of these values are typically changed later.

`void boxssfree(boxssptr boxss);`
`boxssfree` frees a box superstructure, including the boxes.

See also:

`pos2box` - returns a pointer to the box that includes a position.

`setupboxes` - sets up the parameters of a box superstructure.

`assignmolecs` - assigns molecules to boxes.

`boxssoutput` - displays most parameters of a box superstructure.

Reactions

| | |
|---|---|
| <code>typedef struct rxnstruct {</code> | |
| <code>int order;</code> | order of reactions listed: 0, 1, or 2 |
| <code>int *nrxn;</code> | number of reactions for each set of reactants |
| <code>int **table;</code> | lookup list of reaction numbers |
| <code>int lists;</code> | live lists that have reactions |
| <code>int total;</code> | total number of reactions listed |
| <code>char **rname;</code> | names of reactions |
| <code>float *rate;</code> | list of requested reaction rates |
| <code>float *rate2;</code> | reaction rates modified for computation |
| <code>float *rpar;</code> | parameter for reaction of products |
| <code>char *rpart;</code> | type of parameter in rpar |
| <code>int *nprod;</code> | number of products for each reaction |
| <code>moleculeptr **prod; } *rxnptr;</code> | templates of products for each reaction |

rxnstruct (declared in `smollib.h`) is a structure used for a complete set of zeroth order, unimolecular, bimolecular reactions, or higher order reactions. All components of all lists in the structure are owned by the structure. While these structures are complicated, they are also quite versatile and fast to use. The `table` member is a lookup table of all possible reactant combinations, returning an index value of the reaction, if one occurs, called a reaction number. The reaction parameters, such as the reaction names, rates, and product list are then listed sequentially in order of reaction numbers. The dimensionality of the lookup table is the reaction order. If order is 0, then there are no reactants to worry about; `nrxn` and `table` are allocated and initialized so that `nrxn[0]=0` and `table[0]=NULL`, and there are no higher indices allowed. If order is 1, then `nrxn[i]` is the number of unimolecular reactions that molecules of type `i` can undergo and `table[i]` is a list of `nrxn[i]` reaction numbers. For example, `table[i][j]` is the reaction number of the `j`'th unimolecular reaction for molecule `i`. Clearly, empty molecules are included in these lists, accessed with `nrxn[0]` and `table[0]`, where the former should always equal 0 and the latter should always be `NULL`. If order is 2, the same scheme is followed, although now `i` is an index for a two dimensional array. For example, the number of bimolecular reactions possible between molecules `i1` and `i2` is found by first defining `i=nident*i1+i2` with the result of `nrxn[i]` possible reactions. For all reaction orders, `nrxn` and the first index of `table` extend from 0 to `nidentorder`. `lists` is a parameter used to prevent having to scan molecule lists for reactions that don't exist. For zeroth order reactions, `lists` is set to 0. For first order reactions, `lists` is 0 if no molecules have any reactions, 1 if only mobile molecules have reactions, 2 if only immobile molecules have reactions, and 3 for both. For second order reactions, `lists` is 0 if no molecules have any reactions, 1 for only mobile-mobile, 2 for only mobile-immobile, 3 for both.

total is the total number of reactions listed in the structure. rate, rate2, rpar, rpart, nprod, and the first indicies of rname and prod are all allocated to have total elements. The reaction rates are given in rate, although these are generally bulk reaction rates rather than microscopic molecular parameters. rate2 is the rate information used by the simulation routines: if order is 0, rate2 is the average number of molecules produced per time step; if order is 1, rate2 is the probability that a molecule reacts during one time step; if order is 2, rate2 is the squared collision distance between the relevent pair of reactant molecules. Note that the value of rate is independent of the time step, whereas the value of rate2 depends strongly on the time step. Both rate and rate2 are initiallized to -1, and stay that way until they are replaced, if they are replaced. All rate2 values should be replaced and checked by setrates before a simulation is run. rpar is the reversible parameter for the potential reactivity of the products, and can take on any of several meanings, where its type is stored in rpart. See the discussion above for a list of the reversible types and parameters, as well as how they are used.

Following are some code fragments for traversing a reaction structure of arbitrary order. The former fragment walks through the reactions of all reactants and identifies the reaction number for each; the latter one walks through the reactions and identifies the molecule templates for each. To simplify them, the variables order, nident, nrxn, table, total, rate, nprod, and prod have been defined to be equal to the respective elements of a reaction structure or simulation structure.

```
ni2o=intpower(nident,order);
for(i=0;i<ni2o;i++)
    for(j=0;j<nrxn[i];j++)
        r=table[i][j];

for(r=0;r<total;r++)
    for(p=0;p<nprod[r];p++)
        mptr=prod[r][p];
```

```
rxnptr rxnalloc(int order,int nident,int total);
```

rxnalloc allocates and initializes a reaction structure, leaving it fully set up but with zero reactions. It can be used as is, but it won't do anything until reactions are added. order and total need to be at least 0 and nident needs to be at least 1.

```
void rxnfree(rxnptr rxn,int nident);
```

rxnfree frees a reaction structure for any order reaction.

See also:

- findreverserxn - searches reaction structure for reverse reactions.
- setrates - sets the simulation rate2 values, using the macroscopic rate values.
- setproducts - sets initial product separations to account for reversible reactions.
- calcrate - calculates macroscopic rates from microscopic rate2 values.
- loadrxn - loads a reaction structure from a configuration file.
- doreact - actually causes a specified reaction to occur.
- zeroreact - checks and takes care of zeroth order reactions.

unireact - checks and takes care of first order reactions.
 bireact - checks and takes care of second order reactions.
 rxnoutput - displays most parameters of a reaction structure.

Simulation

| | |
|-----------------------------|---|
| typedef struct simstruct { | |
| int dim; | dimensionality of space. |
| int nident; | number of identities, including empty mols. |
| char **name; | names of molecules |
| int graphics; | type of graphics: 0 for none, 1 for opengl |
| int graphicit; | number of time steps per graphics update |
| int framepts; | thickness of frame for graphics |
| float accur; | accuracy, on scale from 0 to 10 |
| float time; | current time in simulation |
| float tmin; | simulation start time |
| float tmax; | simulation end time |
| float dt; | simulation time step |
| rxnptr rxn[3]; | list of reactions |
| molssptr mols; | molecule superstructure |
| wallptr *wlist; | list of walls |
| boxssptr boxes; | box superstructure |
| cmdssptr cmds; | command superstructure |
| float *v1,*v2,*v3; | scratch space, each size dim or nident |
| float *m1,*m2,*m3; | scratch space, each size dim x dim |
| int *z1,*z2,*z3; } *simptr; | scratch space, each size dim or nident |

simstruct (declared in smollib.h) contains and owns all information that defines the simulation conditions, the current state of the simulation, and all other simulation parameters. The scratch space is allocated when the structure is allocated and is for the use of any routine that uses a simulation structure. The v and z scratch space vectors have dimensions that are the larger of dim or nident.

simptr simalloc(int dim,int nident,char *root);
 simalloc allocates a simulation structure. The difc and difm lists are allocated and initialized. Default diffusion matrices are all 0's, except with -1 in the first element. Walls are allocated and inialized. The box superstructure is allocated and initialized, although the list of boxes is left as NULL. The molecule superstructure is left as NULL. The commands superstructure is allocated, but the queue of commands and the output file lists are left as NULLs. root is required for the command superstructure.

void simfree(simpstr sim);
 simfree frees a simulation strucutre, including every part of everything in it.

See also:

loadsimul - loads simulation structure information from configuration file.
 setupstructs - in charge of setting up all structures, including the simulation file.
 simoutput - displays most parameters of a simulation file.

5. Reaction parameter calculation routines – smolib.c

```
int findreverserxn(simptr sim,int i1,int i2,int r,int *optr,int *rptr);
int setrates(simptr sim,int order);
int setproducts(simptr sim,int order,char *erstr);
float calcrate(simptr sim,int i1,int i2,int r,float *pgemptr);
```

These routines work primarily with reaction structures and are intended to be run during program initialization, although after most of the reaction structures have been set up. They convert mass action reaction rates to microscopic simulation parameters and *vice versa*.

```
int findreverserxn(simptr sim,int i1,int i2,int r,int *optr,int *rptr) {
    findreverserxn inputs the reaction defined by reactants i1 and/or i2 and reaction
    number r and looks to see if there is a reverse reaction. If both i1 and i2 are 0, then
    the forward reaction is zeroth order and there is no direct reverse reaction. If either
    i1 or i2 is 0, the forward reaction is first order, and if neither reactant is 0, the
    forward reaction is second order. If there is no reaction number r, an error code of
    -1 is returned. If there is a direct reverse reaction, meaning the products of the
    input reaction are themselves able to react to form identities i1 and i2 (or just one
    of them if the input reaction is first order), then the function returns 1 and the order
    and reaction number of the reverse reaction are pointed to by optr and rptr. If
    there is no direct reverse reaction, but the products of the input reaction are still able
    to react, the function returns 2 and optr and rptr point to the first listed
    continuation reaction. If the products do not react, the function returns 0 (this also
    includes the situation where the products of the input reaction are A and B and there
    is no A+B reaction, although A and/or B can undergo unimolecular reactions, as
    well as all situations in which there are three or more products of a forward
    reaction). Either or both of optr and rptr are allowed to be sent in as NULL values if
    the respective pieces of output information are not of interest.
```

```
int setrates(simptr sim,int order);
    setrates is used to convert the requested reaction rates to values that are useful for
    the simulation routines. Values in the rate element are read and values are written
    to rate2. If a rate element is less than zero, it is assumed to have been unassigned
    and is skipped; in this case, the respective value for rate2 is not modified. For
    zeroth order reactions,  $rate_2$  is the expectation number of molecules that should be
    produced in the entire simulation volume during one time step, which is
 $rate*dt*volume$ . For first order reactions,  $rate_2$  is the probability of a unimolecular
    reaction occurring for an individual reactant molecule during one time step, which is
 $rate/sum*[1-\exp(-sum*dt)]$ , where  $sum$  is the sum of the defined rate values for all
    unimolecular reactions of the reactant. For second order reactions,  $rate_2$  is the
```

squared binding radius of the reactants, found from `bindingradius`. In this case, the reverse parameter is accounted for in the reaction rate calculation if there is a direct reverse reaction and if it is appropriate (see the discussion of “Binding and unbinding radii,” above and the description for `findreverserxn`). This routine also sets the `lists` member of reaction structures. The return value of the function is `-1` for correct operation. If errors occur, which is only possible for illegal inputs (return value of `0`) or bimolecular calculations, the reaction number where the error was encountered is returned. Other than illegal inputs, the only possible errors arise from a diffusion constant of `0` for both reactants, or a directly reversible reaction that has an undeclared reversible type.

```
int setproducts(simptr sim,int order,char *erstr);
```

`setproducts` is used to set the initial separations between reaction products for all reactions of order `order`, based on the corresponding `rpart` character and `rpar` parameter. This is done by calculating the proper separation and then setting the first value of the template molecule `pos` vector to this separation, for all appropriate templates. If `rpart` is either ‘o’ or ‘f’, denoting either randomly oriented offset or fixed orientation offset, then it is assumed that the template molecule position has already been set up; it is not modified again by this routine. Otherwise, it is assumed that the template molecule position vectors have all values equal to `0` initially. If there are illegal inputs, `0` is returned by this routine. If an error occurs, the reaction number where the error was encountered is returned and a message is returned in the string `erstr`, which should have been allocated to size `STRCHAR`; if all assignments work correctly, `-1` is returned and the string is unchanged; and if a reversible reaction was undeclared, `-1` is returned and a warning is returned in the string, although the program does not need to terminate. Possible errors include trying to set product positions for reactions without products, setting relative positions for products in which reactions have only one product, are irreversible, or have multiple reverse reactions, or trying to get a geminate binding probability that is unachievably high due to too long a time step. See the discussion in the section called “Binding and unbinding radii” for more details.

```
float calcrate(simptr sim,int i1,int i2,int r,float *pgemptr);
```

`calcrate` calculates the macroscopic rate constant using the microscopic parameters that have been calculated or that were initially assigned. All going well, these results should exactly match those that were requested initially, although this routine is useful as a check, and for situations where the microscopic values were input rather than the mass action rate constants. For bimolecular reactions that are reversible, the routine calculates rates with accounting for reversibility if the product parameter is `p`, `x`, `r`, `b`, or `?`, and not otherwise. A value of `-1` is returned if input parameters are illegal and a value of `0` is returned if the `rate2` value for the indicated reaction is undefined (`<0`). If reversibility is accounted for and `pgemptr` is not input as `NULL`, `*pgemptr` is set to the probability of geminate recombination of the reactants; otherwise its value is not changed.

6. Initialization routines – smollib.c

```
int loadsimul(simptr *smptr,char *fileroot,char *filename,char *erstr);
int loadrxn(simptr sim,FILE *fptr,int *lctrptr,char *erstr);
boxptr pos2box(float *pos,int dim,boxssptr boxs);
int setupboxes(simptr sim);
int setupstructs(char *root,char *name,simptr *smptr,int vb);
```

Initialization procedures are meant to be called once at the beginning of the program to allocate and set up the necessary structures. These routines call memory allocation procedures as needed.

```
int loadsimul(simptr *smptr,char *fileroot,char *filename,char *erstr);
```

loadsimul loads all simulation parameters from a configuration file, using a format described above. *fileroot* is sent in as the root of the filename, including all colons, slashes, or backslashes; if the configuration file is in the same directory as *Smoldyn*, *fileroot* should be an empty string. *filename* is sent in as just the file name and any extension. *erstr* is sent in as an empty string of size STRCHAR and is returned with an error message if an error occurs. *smptr* is sent in as a pointer to the variable that will point to the simstruct; it is returned pointing to a pointer to an initialized simstruct. This routine calls *loadrxn* to load in any reactions. The following things are set up after this routine is completed: all molecule elements except box; all molecule superstructure elements; all wall elements; box superstructure element *mpbox*, but no other elements; no boxes are allocated or set up; all reaction structure elements except *rate2* and the product template position vectors (*pos* in each product); the command superstructure, including all of its elements; and all simulation structure elements except for sub-elements that have already been listed. If the configuration file loads successfully, the routine returns 0. If the file could not be found, it returns 10 and an error message. If an error was caught during file loading, the return value is 10 plus the line number of the file with an error, along with an error message. If there is an error, all structures are freed automatically.

```
int loadrxn(simptr sim,FILE *fptr,int *lctrptr,char *erstr);
```

loadrxn loads a reaction structure from an already opened disk file pointed to with *fptr*. *lctrptr* is a pointer to the line counter, which is updated each time a line is read. If successful, it returns 0 and the reaction is added to *sim*. Otherwise it returns the updated line counter along with an error message. If a reaction structure of the same order has already been set up, this function can use it and add more reactions to it. It can also allocate and set up a new structure, if needed. If this runs successfully, the complete reaction structure is set up, with the exception of *rate2* and the position vectors of the template molecules, which are all set to 0's (unless the product parameter type is 'o' or 'f', in which case they are set up). If the routine fails, the reaction structure is freed.

```
boxptr pos2box(float *pos,int dim,boxssptr boxs);
```

pos2box returns a pointer to the box that includes the position given in pos, which is a dim size vector. If the position is outside the simulation volume, a pointer to the nearest box is returned. This routine assumes that the entire box superstructure is set up.

```
int setupboxes(simptr sim);
```

setupboxes sets up a superstructure of boxes, and puts things in the boxes, including wall and molecule references. It requires a simulation structure with most things set up, but not box stuff; it's designed for the structure after it's returned from loadsimul. It sets up the box superstructure, then adds indices to each box, then adds the box neighbor list along with neighbor parameters, then adds wall references to each box, and finally creates molecule lists for each box and sets both the box and molecule references to point to each other. The molecule list is $6+3\sqrt{n_{mol}}$ higher than n_{mol} to allow for more molecules; the extra pointers are all set to NULL. The function returns 0 for successful operation and 1 if it was unable to allocate sufficient memory. At the end, all simulation parameters having to do with boxes are set up. However, some lists may still be NULL, if they are empty, where these are `bptr->neigh`, `bptr->wpneigh`, and `bptr->wlist`. Of particular note is that `bptr->wpneigh` is NULL if no neighbors are wrap-around ones, for whatever reason.

```
int setupstructs(char *root, char *name, simptr *smptr, int vb);
```

setupstructs sets up and loads values for all the structures as well as global variables. This routine calls the other initialization routines, so they do not have to be called from elsewhere. Other minor things are set up here, including setting the lookup table for normally distributed random numbers. It also displays the status to stdout and calls output routines for each structure, allowing verification of the initialization. Send in root and name with strings for the path and name of the input file. vb is a flag for verbose operation, 1 for verbose and 0 for quiet. It returns 0 for correct operation and 1 for an error. If it succeeds, `smptr` is returned pointing to a simulation structure. Otherwise, `smptr` is set to NULL and an error message is displayed on stderr.

7. Simulation routines – smolib.c

```
int expandbox(boxptr bptr, int n, int ll);
int assignmolecs(simptr sim, int ll);
void diffuse(simptr sim);
int checkwalls(simptr sim);
int molsort(molssptr mols, int difsort);
int doreact(rxnptr rxn, int r, moleculeptr mptr1, moleculeptr mptr2, simptr sim);
int zeroact(simptr sim);
int unireact(simptr sim);
int bireact(simptr sim, int neigh);
```

```
int expandbox(boxptr bptr, int n, int ll);
```

expandbox is called if it turns out that a box was not allocated with enough space for molecules. `bptr` is a pointer to a box that needs expanding, `n` is the number of

additional molecule spaces to add, and `ll` is the live list to expand. If `n` is negative, the box is shrunk and any molecule pointers that no longer fit are simply left out. The book keeping elements of the box are updated. The function returns 0 if it was successful and 1 if there was not enough memory for the request.

```
int assignmolecs(simptr sim,int ll);
```

`assignmolecs` puts molecules in boxes by overwriting the lists of molecules that are in each box with molecules from `mols`. It only assigns the live list number `ll`. Molecules that are outside the set of boxes are assigned to the nearest box. If more molecules belong in a box than actually fit, 5 more spaces are allocated using `expandbox`. The function returns 0 unless memory could not be allocated by `expandbox`, in which case only some of the molecules are assigned and it returns 1.

```
void diffuse(simptr sim);
```

`diffuse` does the diffusion for all molecules in the `live[0]` list (mobile), over one time step. Walls are ignored and molecules are not reassigned to the boxes. If there is a diffusion matrix, it is used for anisotropic diffusion; otherwise isotropic diffusion is done, using the `difstep` parameter.

```
int checkwalls(simptr sim);
```

`checkwalls` does the reflection, wrap-around, or absorption of molecules at walls by checking the current position, relative to the wall positions (as well as a past position for absorbing walls). It does not reassign the molecules to boxes or sort the live and dead ones, although this typically will not be required anyhow. It returns the number of wall collisions that were detected and processed during that time step.

```
int molsort(molssptr mols,int difsort);
```

`molsort` updates the live and dead lists of both a molecule superstructure and the relevant boxes after a reaction or other changes. If `difsort` is 0, it assumes that live molecules are in the correct live list, otherwise, it sorts this too (the only way it's likely to need sorting is if a command changes a molecule's identity). First it might deal with diffusion sorting by moving missorted molecules to the resurrected list. Afterwards, and in normal operation, it moves the resurrected molecules off the top of the dead list (those numbered between `top` and `nd-1`) to the live list. Then it moves the expired molecules from the live list to the dead list. Finally it compacts the live list. Molecule ordering in lists is not preserved. It is required that the box numbers for molecules are correct and that the molecule is listed in the box list corresponding to the master live lists. The routine returns 0 for normal operation and 1 if memory could not be allocated when a box was being expanded.

```
int doreact(rxnptr rxn,int r,moleculeptr mptr1,moleculeptr mptr2,simptr sim);
```

`doreact` executes a reaction that has already been determined to have happened. `rxn` is the reaction, `r` is the reaction number and `mptr1` and `mptr2` are the reactants, where `mptr2` is ignored for unimolecular reactions, and both are ignored for zeroth order reactions. Reactants are killed, but left in the live lists. Any products are created on the dead list, for transfer to the live list by the `molsort` routine.

Molecules that are created are put at the reaction position, which is the average position of the reactants weighted by the inverse of their diffusion constants, plus an offset from the product definition. The cluster of products is typically rotated to a random orientation. If the displacement was set to all 0's (recommended for non-reacting products), the routine is fairly fast, putting all products at the reaction position. If the rpart character is 'f', the orientation is fixed and there is no rotation. Otherwise, a non-zero displacement results in the choosing of random angles and vector rotations. If the system has more than three dimensions, only the first three are randomly oriented, while higher dimensions just add the displacement to the reaction position. The function returns 0 for successful operation and 1 if more molecules are required than were initially allocated.

```
int zeroreact(simptr sim);
```

zeroreact figures out how many molecules to create for each zeroth order reaction and then tells doreact to create them. It returns the number of molecules created, or -1 if not enough molecules were allocated initially.

```
int unireact(simptr sim);
```

unireact determines whether unimolecular reactions occurred, considering both live lists. Reactions that do occur are sent to doreact to process them. The function returns the number of reactions that occurred during that time step, or -1 if not enough molecules were allocated initially.

```
int bireact(simptr sim,int neigh);
```

bireact determines whether bimolecular reaction occurred, sending ones that do occur to doreact. neigh tells the routine whether to consider only reactions between neighboring boxes (neigh=1) or only reactions within a box (neigh=0). The former are relatively slow and so can be ignored for qualitative simulations by choosing a lower simulation accuracy value. In cases where walls are periodic, it is possible to have reactions over the system walls. The function returns the number of reactions that occurred during that time step, or -1 if not enough molecules were allocated initially.

A good sequence for using these routines is: cmdcheck, diffuse, assignmolecs, checkwalls, zeroreact, unireact, bireact, molsort, checkwalls. The reason for checking walls after reactions, as well as before, is that products may be created outside the simulation volume. This doesn't really matter, except that commands might care and it doesn't look good in a graphical output. If commands are really picky about molecules being in the correct boxes, it is also possible to call assignmolecs a second time too. The recommended sequence was changed for version 1.51 to put assignmolecs before checkwalls, with the logic that wall checking assumes that molecules are in the correct boxes, whereas the reactions are less sensitive. In most situations, the slowest routine is expected to be bireact, although essentially all of the routines have to do a significant amount of scanning through box lists and molecule lists.

8. Output routines – smolib.c

```
void simoutput(simptr sim);
void walloutput(int dim,wallptr *wlist);
void molssoutput(simptr sim);
void boxoutput(int dim,boxssptr boxs);
void boxssoutput(simptr sim);
void rxnoutput(simptr sim,int order);
void checkparams(simptr sim);
```

These output routines send program parameters and diagnostics to the standard output. This is useful for making sure that the program is working as expected and that the internal parameters have physically sensible values.

```
void simoutput(simptr sim);
    simoutput prints out the overall simulation parameters, including simulation time
    information, graphics information, the number of dimensions, what the molecule
    types are, the output files, and the accuracy.

void walloutput(int dim,wallptr *wlist);
    walloutput prints the wall structure information, including wall dimensions,
    positions, and types, as well as the total simulation volume.

void molssoutput(simptr sim);
    molssoutput prints all the parameters in a molecule superstructure. While it should
    not be needed, hopefully, this routine looks for and prints out information on
    molecules that are not sorted correctly in the live and dead lists. It also prints out
    information about each molecule, including diffusion constants, rms step lengths,
    colors, and display sizes.

void boxoutput(int dim,boxssptr boxs);
    boxoutput simply lists every virtual box, along with all the details about it, where
    these details are the index, the number of neighbors, the neighbor mid-point, the
    number of maximum number of molecules, what the neighbors are, and what the
    wrapping codes are. As the program is currently written, this function is never
    called, although it could be to look for errors in box setting up.

void boxssoutput(simptr sim);
    boxssoutput displays statistics about the box superstructure, including total number
    of boxes, number on each side, dimensions, and the minimum position. It also
    prints out the requested and actual numbers of molecules per box.

void rxnoutput(simptr sim,int order);
    rxnoutput displays the complete contents of a reaction structure for order order. It
    also does some other calculations, such as the probability of geminate reactions for
    the products and the diffusion and activation limited rate constants.

void checkparams(simptr sim);
```

checkparams checks that the simulation parameters, including parameters of sub-structures, have reasonable values. If values seem to be too small or too large, a warning is displayed to the standard output, although this does not affect continuation of the program.

9. Command interpreter routines – smollib2.c

Declaration in smollib2.h:

```
int docommand(void *cmdfnarg,cmdptr cmd,char *line);
```

Declarations in smollib2.c:

```
int cmdstop(simptr sim,cmdptr cmd,char *line2);
int cmdpause(simptr sim,cmdptr cmd,char *line2);
int cmdoverwrite(simptr sim,cmdptr cmd,char *line2);
int cmdincrementfile(simptr sim,cmdptr cmd,char *line2);
int cmdifno(simptr sim,cmdptr cmd,char *line2);
int cmdifless(simptr sim,cmdptr cmd,char *line2);
int cmdifmore(simptr sim,cmdptr cmd,char *line2);
int cmdpointsource(simptr sim,cmdptr cmd,char *line2);
int cmdkillmol(simptr sim,cmdptr cmd,char *line2);
int cmdequilmol(simptr sim,cmdptr cmd,char *line2);
int cmdreplacexyzmol(simptr sim,cmdptr cmd,char *line2);
int cmdmodulatemol(simptr sim,cmdptr cmd,char *line2);
int cmdreact1(simptr sim,cmdptr cmd,char *line2);
int cmdmolcount(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols2(simptr sim,cmdptr cmd,char *line2);
int cmdlistmols3(simptr sim,cmdptr cmd,char *line2);
int cmdmolpos(simptr sim,cmdptr cmd,char *line2);
int cmdmolmoments(simptr sim,cmdptr cmd,char *line2);
int cmdsavesim(simptr sim,cmdptr cmd,char *line2);
int cmdexcludebox(simptr sim,cmdptr cmd,char *line2);
int cmdexcludesphere(simptr sim,cmdptr cmd,char *line2);
int cmdincludeecoli(simptr sim,cmdptr cmd,char *line2);

int insideecoli(float *pos,float *ofst,float rad,float length);
void putinecoli(float *pos,float *ofst,float rad,float length);
```

Command strings are not parsed, checked, or even looked at during simulation initialization. Instead, they are run by the command interpreter during the simulation. Command routines are given complete freedom to look at and/or modify any part of a simulation structure or sub-structure. This, of course, also gives commands the ability to crash the computer program, so they need to be written carefully to prevent this. Every command is sent a pointer to the simulation structure in sim, as well as a string of command parameters in line2.

```
int docommand(void *cmdfnarg,cmdptr cmd,char *line);
```

docommand is given the simulation structure in `sim`, the command to be executed in `cmd`, and a line of text which includes the entire command string. It parses the line of text only into the first word, which specifies which command is to be run, and into the rest of the line, which contains the command parameters. The rest of the line is then sent to the appropriate command routine as `line2`. The return value of the command that was called is passed back to the main program from `docommand`. These routines return 0 for normal operation, 1 for an error that does not require simulation termination, 2 for an error that requires simulation termination, and 3 for a time step termination but no simulation termination (for pausing).

`int cmdstop(simptr sim,cmdptr cmd,char *line2);`
`cmdstop` returns a value of 2, meaning that the simulation should stop. Any contents of `line2` are ignored.

`int cmdpause(simptr sim,cmdptr cmd,char *line2);`
`cmdpause` causes the simulation to pause until the user tells it to continue. Continuation is effected by either pressing the space bar, if OpenGL is used for graphics, or by pressing enter if output is text only. The return value is 0 for non-graphics and 3 for graphics. Any contents of `line2` are ignored.

`int cmdoverwrite(simptr sim,cmdptr cmd,char *line2);`
`cmdoverwrite` overwrites a prior output file. See the user manual.

`int cmdincrementfile(simptr sim,cmdptr cmd,char *line2);`
`cmdincrementfile` closes a file, increments the name and opens that one for output. See the user manual.

`int cmdifno(simptr sim,cmdptr cmd,char *line2);`
`cmdifno` reads the first word of `line2` for a molecule name and then checks the appropriate simulation live list to see if any molecules of that type exist. If so, it does nothing, but returns 0. If not, it sends the remainder of `line2` to `docommand` to be run as a new command, and then returns 0. It returns 1 if the molecule name was missing or not recognized.

`int cmdifless(simptr sim,cmdptr cmd,char *line2);`
`cmdifless` is identical to `cmdifno`, except that it runs the command in `line2` if there are less than a listed number of a kind of molecules in the appropriate live list.

`int cmdifmore(simptr sim,cmdptr cmd,char *line2);`
`cmdifmore` is identical to `cmdifno` except that it runs the command in `line2` if there are more than a listed number of a kind of molecules in the appropriate live list.

`int cmdpointsource(simptr sim,cmdptr cmd,char *line2);`
`cmdpointsource` reads `line2` for a molecule name, followed by the number of molecules that should be created, followed by the `dim` dimensional position for them. If all reads well, it creates the new molecules in the system at the appropriate position. They are added to the dead list and then the lists are sorted.

```

int cmdkillmol(simptr sim,cmdptr cmd,char *line2);
    cmdkillmol reads line2 for a molecule name and then kills all molecules of that
    name from the appropriate live list by setting their identities to 0. The molecule
    lists are then sorted.

int cmdequilmol(simptr sim,cmdptr cmd,char *line2);
    cmdequilmol equilibrates a pair of molecular species, allowing the efficient
    simulation of rapid reactions. It reads two molecule names from line2, followed by
    a probability value. Then, it looks for all molecules in the live lists with either of
    the two types and replaces them with the second type using the listed probability or
    with the first type using 1– the listed probability.

int cmdreplacexyzmol(simptr sim,cmdptr cmd,char *line2);
    cmdreplacexyzmol reads the name of a molecule following by a dim dimensional
    point in space from line2. Then, it searches the fixed live list for any molecule that
    is exactly at the designated point. If it encounters one, it is replaced by the listed
    molecule, and then the live lists are sorted if appropriate. This routine stops
    searching after one molecule has been found, and so will miss additional molecules
    that are at the same point.

int cmdmodulatemol(simptr sim,cmdptr cmd,char *line2);
    cmdmodulatemol is identical to cmdequilmol except that the equilibration probability
    is not fixed, but is a sinusoidally varying function. After reading two molecule
    names from line2, this routine then reads the cosine wave frequency and phase shift,
    then calculates the probability using the function prob=0.5*(1.0-cos(freq*sim-
    >time+shift)).

int cmdreact1(simptr sim,cmdptr cmd,char *line2);
    cmdreact1 reads line2 for the name of a molecule followed by the name of a
    unimolecular reaction. Then, every one of that type of molecule is caused to
    undergo the listed reaction, thus replacing each one by reaction products.
    Molecules are sorted at the end. This might be useful for simulating a pulse of
    actinic light, for example.

int cmdmolcount(simptr sim,cmdptr cmd,char *line2);
    cmdmolcount reads the output file name from line2. Then, to this file, it saves one
    line of text listing the current simulation time, followed by the number of each type
    of molecule in the system. This routine does not affect any simulation parameters.

int cmdlistmols(simptr sim,cmdptr cmd,char *line2);
    cmdlistmols reads the output file name from line2. To this file, it saves a list of
    every individual molecule in both live lists of the simulation, along with their
    positions. This routine does not affect any simulation parameters.

int cmdlistmols2(simptr sim,cmdptr cmd,char *line2);

```

`cmdlistmols2` reads the output file name from `line2`. To this file, it saves the number of times this command was invoked using the `invoke` element of commands, a list of every individual molecule in both live lists of the simulation, along with their positions. This routine does not affect any simulation parameters. Routine originally written by Karen Lipkow and then rewritten by me.

```
int cmdlistmols3(simptr sim,cmdptr cmd,char *line2);
```

`cmdlistmols3` reads a molecule name and the output file name from `line2`. To this file, it saves the number of times the command was invoked, the identity of the molecule specified, and the positions of every molecule of the specified type. This routine does not affect any simulation parameters.

```
int cmdmolpos(simptr sim,cmdptr cmd,char *line2);
```

`cmdmolpos` reads a molecule name and then the output file name from `line2`. To this file, it saves one line of text with the positions of each molecule of the listed identity. This routine does not affect any simulation parameters.

```
int cmdmolmoments(simptr sim,cmdptr cmd,char *line2);
```

`cmdmolmoments` reads a molecule name and then the output file name from `line2`. To this file, it saves in one line of text: the time and the zeroth, first, and second moments of the distribution of positions for all molecules of the type listed. The zeroth moment is just the number of molecules (of the proper identity); the first moment is a *dim* dimensional vector for the mean position; and the second moment is a *dimxdim* matrix of variances. This routine does not affect any simulation parameters.

```
int cmdsavesim(simptr sim,cmdptr cmd,char *line2);
```

`cmdsavesim` reads the output file name from `line2` and then saves the complete state of the system to this file, as a configuration file. This output can be run later on to continue the simulation from the point where it was saved.

```
int cmdexcludebox(simptr sim,cmdptr cmd,char *line2);
```

`cmdexcludebox` allows a region of the simulation volume to be effectively closed off to molecules. The box is defined by its low and high corners, which are read from `line2`. Any molecule, of any type, that entered the box during the last time step, as determined by its `pos` and `posx` structure members, is moved back to its previous position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations. This routine ought to be replaced with a proper treatment of surfaces in the main program (rather than with interpreter commands), but that's a lot more difficult.

```
int cmdexcludesphere(simptr sim,cmdptr cmd,char *line2);
```

`cmdexcludesphere` is like `cmdexcludebox` except that it excludes a sphere rather than a box. The sphere is defined by its center and radius, which are read from `line2`. Any molecule, of any type, that entered the sphere during the last time step, as determined by its `pos` and `posx` structure members, is moved back to its previous

position. This is not the correct behavior for a reflective surface, but is efficient and expected to be reasonably accurate for most situations.

```
int cmdincludeecoli(simptr sim,cmdptr cmd,char *line2);
```

`cmdincludeecoli` is the opposite of the `excludebox` and `excludesphere` commands. Here, molecules are confined to an *E. coli* shape and are put back inside it if they leave. See the user manual for more about it. Unlike the other rejection method commands, this one works even if a molecule was in a forbidden region during the previous time step; in this case, the molecule is moved to the point on the *E. coli* surface that is closest. Because of this difference, this command works reasonably well even if it is not called at every time step.

```
int insideecoli(float *pos,float *ofst,float rad,float length);
```

This is a short utility routine used by the command `cmdincludeecoli`. It returns a 1 if a molecule is inside an *E. coli* shape and a 0 if not. `pos` is the molecule position, `ofst` is the physical location of the cell membrane at the center of the low end of the cell (the cell is assumed to have its long axis along the *x*-axis), `rad` is the cell radius used for both the cylindrical body and the hemispherical ends, and `length` is the total cell length, including both hemispherical ends.

```
void putinecoli(float *pos,float *ofst,float rad,float length);
```

This is another short utility routine used by the command `cmdincludeecoli`. It moves a molecule from its initial position in `pos` to the nearest surface of an *E. coli* shape. Parameters are the same as those for `insideecoli`.

10. Simulation management routines and main() – smoldyn.c

Declarations in `smoldyn.c` that do not require OpenGL:

```
int simulatetimestep(simptr sim,int ctr[]);
void endsimulate(simptr sim,int vb,int ctr[],time_t tstt,int er);
void smolsimulate(simptr sim,int vb);
int main(int argc,char *argv[]);
```

Declarations in `smoldyn.c` that require OpenGL:

```
void RenderScene(void);
void TimerFunction(int value);
void smolsimulategl(simptr sim,int vb);
```

The source code file `smoldyn.c` contains high level functions that allow program entry from the shell, exit to the shell, functions that manage the simulation, and functions that take care of graphics. These functions are all declared locally and thus cannot be called from externally. The structure of this segment is largely determined by the constraints of the OpenGL framework, in which control is passed from the main program to OpenGL and is never returned. As a result, the only way to quit a simulation that uses

graphics is either by having the user choose quit from the menu or with the standard library command `exit(0)`. The former method was chosen, but has the drawback that there is no way to free the simulation structure before terminating the program. Instead, *Smoldyn* relies on the system to free the allocated memory. Without graphics, a more conventional C structure is used, including freeing of memory upon completion and a normal return to the shell, although the structure of the code is still slightly strange due to its need to be compatible with the OpenGL segment.

Routines in which OpenGL is required

`void RenderScene(void);`

`RenderScene` is the call-back function for OpenGL that displays the graphics. This function simply draws a box for the simulation volume, as well as points for each molecule. It uses the global `Sim` variable.

`void TimerFunction(int value);`

`TimerFunction` is the call-back function for OpenGL that runs the simulation. It always tells OpenGL to update the graphics and to call back. If `value` is 0 and the simulation has not been paused, this means that the simulation is continuing, so this routine simulates one or more simulation time steps (the value depends on `sim->graphicit`). Otherwise, it does not do any computations because either the simulation has been paused (`gl2State` equal to 1) or the simulation is complete (`value` equal to 1). In these cases, the user can manipulate the graphics or quit the program. When the user quits the program, the event is captured by OpenGL, which then transfers control directly to the shell. A modification new to version 1.52 is that if `gl2State` returns 2, this indicates that the user pressed 'Q' to indicate that the simulation should stop, as though it had completed normally.

`void smolsimulategl(simptr sim,int vb);`

`smolsimulategl` starts the simulation using OpenGL graphics. It does all OpenGL initializations, registers OpenGL call-back functions, sets the global variables to their proper values, and then hands control over to OpenGL. This function does not return.

No OpenGL required

`int simulatetimestep(simptr sim,int ctr[]);`

`simulatetimestep` runs the simulation over one time step. If an error is encountered at any step, or a command tells the simulation to stop, or the simulation time becomes greater than or equal to the requested maximum time, the function returns an error code to indicate that the simulation should stop; otherwise it returns 0 to indicate that the simulation should continue. Error codes are 1 for simulation completed normally, 2 for error with `assignmolecs`, 3 for error with `zeroreact`, 4 for error with `unireact`, 5 for error with `bireact`, 6 for error with `molsort`, or 7 for terminate instruction from `docommand` (e.g. stop command). Errors 2 and 6 arise from insufficient memory when boxes were being expanded and errors 3, 4, and 5 arise from too few molecules being allocated initially.

`void endsimulate(simptr sim,int vb,int ctr[],time_t ttt,int er);`
endsimulate takes care of things that should happen when the simulation is complete. This includes executing any commands that are supposed to happen after the simulation, displaying numbers of simulation events that occurred, and calculating the execution time. `er` is a code to tell why the simulation is ending, which has the same values as those returned by `simulatetimestep`. If graphics are used, this routine just returns to where it was called from (which is `TimerFunction`); otherwise, it frees the simulation structure and then returns (to `smolsimulate` and then `main`).

`void smolsimulate(simptr sim,int vb);`
`smolsimulate` runs the simulation without graphics. It does essentially nothing other than running `simulatetimestep` until the simulation terminates. At the end, it calls `endsimulate` and returns.

`int main(int argc,char *argv[]);`
`main` is a simple routine that provides an entry point to the program. It checks the command line arguments, prints a greeting, inputs the configuration file name from the user, and then calls `setupstructs` to load the configuration file and set up all the structures. If all goes well, it calls `simulate` or `simulategl` to run the simulation. As OpenGL never returns control to the main program, the exit point of *Smoldyn* is only here is OpenGL is not used.

11. *Smoldyn* modifications

Modifications made for version 1.5 (released 7/03).

Added heirarchical configuration file name support.

Zeroreact assigns the correct box for new molecules.

The user can choose the level of detail for the bimolecular interactions (just local, nearest neighbor, all neighbor, including periodic, etc.)

Bimolecular reactions were slow if most boxes are empty. Solution was to go down molecule list rather than box list.

Absorbing wall probabilities were made correct to yield accurate absorption dynamics at walls.

Cleaned up and got rid of old commands.

The current time input was made useful.

Graphics were improved by adding perspective and better user manipulation.

Simulation pausing was made possible using graphics and improved without graphics.

If a command was used with a wrong file name, the command string became corrupted during the final command call. This was fixed by Steve Lay.

Fixed the neighbor list for bimolecular reactions between mobile and immobile reactants.

Reactions were made possible around periodic boundaries.

Molecules were lost sometimes. This bug was fixed: 4 lines before end of `molsort`:

```
was: while(!live[m]) {  
now: while(!live[m]&&mol[1]) {
```

Output files now allow the configuration file to be in a different folder as *Smoldyn*.

Added an output file root parameter.

Added the command `replacexyzmol`. Afterwards, the code for the command was sped up considerably.

Sped up the command `excludebox`.

Command time reports were fixed for type `b` and `a` commands.

Added more types of command timing codes.

Improved accuracy of `unireact` so that it correctly accounts for multiple reactions from one identity.

Improved product parameter entry and calculation, as well as the output about reaction parameters.

Added the routine `checkparams` to check that the simulation parameters are reasonable.

Modifications made for version 1.51 (released 9/5/03).

Fixed a minor bug in `doreact` which allowed the molecule superstructure indices to become illegal if not enough molecules were allocated.

Fixed a minor bug in `cmdreact1` which did not check for errors from `doreact`.

Added command `molpos`.

Moved version number from a `printf` statement to a macro, in `smoldyn.c` file.

Added command `listmols2`, from a file sent to me by Karen Lipkow.

Fixed a minor bug in `checkparams` that printed warnings for unused reactions.

In `simulatetimestep` in `smoldyn.c`, the order of operations was `diffuse`, `checkwalls`, and then `assignmolecules`. The latter two were swapped, which should make wall checking more accurate when time steps are used that are so long that rms step lengths are a large fraction of box sizes. The new version is less accurate than before when the simulation accuracy is less than 10, but should be more accurate when it is 10.

Replaced the `coinrand` call in `unireact`, which determines if a reaction occurred, with `coinrand30` to allow better accuracy with low probabilities. Also changed the relevant check in `checkparams`.

Improved reactive volume test in `checkparams`.

Increased `RANDTABLEMAX` from 2047 to 4095.

Some modifications were made to `random.h`.

Fixed a major bug in `rxnfree`, regarding the freeing of the table elements.

Modifications made for version 1.52 (released 10/24/03)

Changed comments in `rxnparam.h` and `rxnparam.c`, but no changes in code.

Changed `cmdsavesim` in `smolib2.c` to allow it to compile with `gcc`.

Added another call to `assignmolecules` in `simulatetimestep` in `smoldyn.c`, after the call to `checkwalls`, to make sure that all molecules are assigned properly before checking reactions. This slows things down some, but should allow slightly longer time steps.

To the `opengl2.c` file, the `KeyPush` function was modified so now pressing 'Q' sets the `Gl2PauseState` to 2, to indicate that a program should quit. A few modifications were also made in `smoldyn.c` function `TimerFunction` to make use of this.

Corrected two significant bugs in the `checkwalls` function in `smollib.c` regarding absorbing walls. First, it didn't work properly for low side walls. Also, the probability equation was incorrect, which was noticed by Dan Gillespie.

Fixed a minor bug in `cmdsavesim` in `smollib2.c` file, which caused an output line for `rate_internal` to be displayed for declared but unused reactions.

Several commented out functions in `loadrxn` were removed because they were obsolete and have been replaced by `product_param`. They were: `p_gem`, `b_rel`, `b_abs`, `offset`, `fixed`, and `irrev`.

A command superstructure was created, which moved several structure elements out of the simulation structure. No new functionality was created, but the code is cleaner now. New routines are `cmdssalloc` and `cmdssfree`. Updated routines are: `simalloc`, `simfree`, `loadsimul`, `setupstructs`, `cmdoutput` (including function declaration), `openoutputfiles` (including function declaration and ending state if an error occurs), `commandpop` (including function declaration), `checkcommand`, `endsimulate`, `savesim`, `main`, and all commands that save data to files.

Renamed the "test files" folder to "test_files".

Modifications made for version 1.53 (released 2/9/04)

Cleaned up commands a little more by writing routine `getfptr` in `smollib2.c` and calling it from commands that save data, rather than repeating the code each time.

All routines that dealt with the command framework were moved to their own library, called `SimCommands`. This also involved a few function name and argument changes, affecting `smoldyn.c`, `smollib.c`, `smollib.h`, `smollib2.c`, and `smollib2.h`.

Formatting was cleaned up for structure output routines.

Swapped drawing of box and molecules, so box is on top. Also increased default box line width to 2 point.

Computer now beeps when simulation is complete.

Modified `SimCommand` library so that each invocation of a command is counted and also changed declaration for `docommand` in `smollib2`. This change was useful for improving the command `listmols2` so it can be run with several independent time counters. Also, wrote command `listmols3`.

Wrote the new configuration file statement `boxsize`.

Wrote the new commands `excludesphere` and `includeecoli`.

Wrote the commands `overwrite` and `incrementfile`, which also involved some changes to the `SimCommand` library and required the new configuration file statement `output_file_number`.

Added a new configuration file statement `frame_thickness`.

When simulation is paused using OpenGL, the simulation time at which it was paused is now displayed to the text window.

Modifications made for version 1.54 (released 3/3/04)

Swapped order of commands and OpenGL drawing so that commands are executed before displaying results. Also wrote section 3.2 of the documentation to discuss this ordering and other timing issues.

Wrote documentation section 3.3 on surface effects on reaction rates and added the *reactW* set of test files.

The wish list

Addition of internal surfaces. There is a tremendous amount that would be very nice here. The simplest addition would be simple reflective surfaces. Next are semi-permeable surfaces, non-diffusing membrane-bound molecules, diffusing membrane-bound molecules, moveable surfaces, etc. Similarly, it would be nice to have fibers (such as DNA), fiber-bound molecules, etc.

Variable simulation time steps. The less difficult method would be to figure out how fast the system is changing as a whole and to adjust the simulation time step to compensate for this. A challenge though, is that it is nearly impossible to redo a step that was determined to have been too long without introducing significant statistical bias. The more difficult method is to use different length time steps for different molecules, so as to poll labile ones more often than stable ones.

More functionality for the runtime command interpreter. It would be nice if commands could communicate with each other, have their own storage space, etc. An idea for this is to establish a bulletin board within the command superstructure, on which commands could post and read memos. Also, it would be nice to avoid round-off errors when using 'e' or 'n' timing codes; similarly, 'n' should work properly with non-uniform time steps.

Ability to store graphics as a TIFF, Quicktime movie or in some other standard format.

More graphics manipulations, such as panning, and changing viewing position.

Configuration file compatibility with SBML, XML, or other standards. Also, inclusion of *Smoldyn* into *BioSpice*.

Inclusion of continuous concentrations for chemical species that are abundant.